Appeared in the proceedings of the *13-th Conference on Foundations of Software Technology and Theoretical Computer Science* (FSTTCS'93), 15–17 December 1993, Bombay, India, pp. 41–51, LNCS 761, Springer-Verlag, Berlin.

Conventional and Uniqueness Typing in Graph Rewrite Systems

Extended abstract

Erik Barendsen Sjaak Smetsers
University of Nijmegen*

Abstract

In this paper we describe a Curry-like type system for graphs and extend it with *uniqueness* information to indicate that certain objects are only 'locally accessible'. The correctness of type assignment guarantees that no external access on such an object will take place in the future. We prove that types are preserved under reduction (for both type systems) for a large class of rewrite systems. Adding uniqueness information provides a solution to two problems in implementations of functional languages: efficient space management and interfacing with non-functional operations.

1. Introduction

There are several models of computation that can be viewed as a basis for functional programming. Traditional examples are the *lambda calculus* and *term rewrite systems*. Graph rewriting is a relatively new concept, providing a model that is sufficiently elegant and abstract, but at the same time incorporates mechanisms that are more realistic with respect to actual implementation techniques.

Graph rewrite systems were introduced in Barendregt *et al.* [1987a]. The present paper deals with a restricted form of GRS's: the so-called *term graph rewrite systems* (TGRS's, see Barendregt *et al.* [1987b] and Barendsen & Smetsers [1992]). TGRS's are very well suited as a basis for (implementation of) functional languages, as is demonstrated by the graph rewrite language *Concurrent Clean*, see Nöcker *et al.* [1991].

Conventional types

The concept of typing in lambda calculus is well known. To study the effect of *patterns* in function definitions on typing, a Curry-like type assignment system

*Computing Science Institute, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands, e-mail erikb@cs.kun.nl, sjakie@cs.kun.nl, fax +31.80.652525.

on (applicative) term rewriting systems has been developed by Bakel *et al.* [1992].

In the present paper the notion of type assignment to TRS's of Bakel *et al.* [1992] will be extended to general term graphs (i.e. graphs that are not necessarily trees) in a very natural way. Some aspects of typing are even more convenient in graph theoretical setting. E.g. special contexts for variables are not necessary since multiple occurrences of the same variable are represented by one single node. Moreover, the extra features of graph rewriting (shared c.q. cyclic objects) are treated without extra effort.

As in Bakel *et al.* [1992], type assignment is not defined using a set of deduction rules but, more directly, by supplying nodes of the graphs and of the rewrite rules with types in a consistent way. The consistency is expressed in a 'local' constraint for each node. Graph symbols are supplied with a type by a so called *type environment*.

Uniqueness types

The underlying motivation for uniqueness types was given by two fundamental problems in practical functional programming and the implementation of functional languages using sharing techniques.

The first problem is the *space behaviour* of functional programs during execution. In a reduction step one often has to construct complicated structures, involving creation of new nodes. One could improve the efficiency of the implementation by re-using the space of obsolete objects of the part of the graph being rewritten, thus performing garbage collection on the spot. It would even be better if one could predict *at compile time* which arguments of a function will become garbage during rewriting. This is called *compile time garbage collection*. This is often the only way to handle complex data structures efficiently.

A second issue is the incorporation of essentially nonfunctional operations in the formalism of graph rewriting, e.g. for dealing with input-output. File updating, for example, is an operation with side-effects, possibly disturbing referential transparency. Such operations are safe, however, if there exists only a single reference to the object being modified, at the moment the modification takes place.

The technique presented here involves incorporation of *locality* or *uniqueness* information in the type system mentioned above. Types are therefore extended with so called uniqueness attributes.

In the type of a function F it can now be indicated that a specific argument should be 'unique'. The intended meaning is that at the moment of evalutation, the corresponding object is local for F, i.e. can only be accessed via F. The type system will allow only applications of F of this kind.

This locality information can be used to solve the problems indicated above. Suppose a function argument is unique. If (part of) the argument is not used in the result, it can be concluded that this part becomes garbage in any concrete application. It is then possible to re-use the space or even the contents of obsolete objects when building the result. The second problem is solved by typing the 'dangerous' updating operations in such a way that they require unique arguments.

In order to achieve this, the notion of type assignment (for conventional typing) is extended: the type correctness of an application FX depends not only on the argument type of F and the type of X, but also on the way X is passed to F: if F expects a 'unique' argument then X should only be accessible by F, e.g. by requiring a reference count of 1. This straightforward reference count approach is rather rough. In practice one usually has a specific evaluation order in mind. In this paper we present a more liberal analysis using this information. The correctness of a function application now depends on the demanded argument type, the offered argument type and the label of the reference to that argument. This dependency is formulated in terms of type *coercions*. We prove that uniqueness typing is preserved under graph rewriting, for a sufficiently large class of graph rewrite systems.

The weighted reference count analysis, as presented in section 4, is inspired by Guzmán & Hudak [1990]. This paper addresses the mutabilty problem using a 'single threaded polymorphic lambda calculus' (*poly*-λ_{st}). It uses the operational semantics of lambda-graph reduction of Wadsworth [1971]. In our paper the analysis is performed in the formalism of graph rewriting, which is obviously more direct. The effect of cyclic structures (not present in the paper mentioned above) and general pattern matching are studied in the general graph rewriting setting presented here. In Wadler [1990], a type system including linear types is developed. The paper also used Wadsworth's lambda reduction. The coercions described in our paper are implicit in Wadler [1990], in a much more restricted form. The ideas presented here inspired Jacobs [1993] to develop a *logical* system explicitly mixing conventional and linear constructive logic. The approach described here is likely to offer a 'propositions-as-types' notion. A combination of conventional and uniqueness typing has been incorporated in the lazy functional graph rewriting language *Concurrent Clean*. So far, it has been used for the implementation of arrays and of an efficient high-level library for screen and file I/O (see Achten *et al.* [1993]).

2. Graph rewriting

Term graph rewrite systems were introduced in Barendregt *et al.* [1987b]. This section summarizes some basic notions for (term) graph rewriting as presented in Barendsen & Smetsers [1992].

The objects of our interest are directed graphs in which each node has a specific label. The number of outgoing edges of a node is determined by its label. In the sequel we assume that $\mathcal{N}$ is some basic set of *nodes* (infinite; one usually takes $\mathcal{N} = \mathbb{N}$), and Σ is a set of *symbols* with *arity* in $\mathbb{N}$.

A *labeled graph* (over $\langle \mathcal{N}, \Sigma \rangle$) is a triple $g = \langle N, \mathit{symb}, \mathit{args} \rangle$ where $N \subseteq \mathcal{N}$ is a set of nodes. Furthermore, *symb* and *args* are functions respectively indicating the label (the *symbol*) and the outgoing references (the *arguments*) of each node in g. The i-th argument of n is denoted by $\mathit{args}(n)_i$; the combination (n, i) is called a *reference* (of g). A graph is called *rooted* if some node $r \in N$ is explicitly indicated as *root*. If g is a (rooted) graph, then its components are referred to as N_g, symb_g, args_g (and r_g) respectively.

A *path* in g is a sequence p of successive references of g. We use $p : n \rightsquigarrow m$ to indicate that p leads from node n to m. If such path exists, we say that *m is reachable from n*. The *subgraph of g at n* (notation $g \mid n$) is the rooted graph that is obtained from g be taking n as root and removing all the nodes not reachable from n.

Term graph rewrite systems

Rewrite rules specify transformations of graphs. Each rewrite rule R is a triple $\langle g, l, r\rangle$ consisting of a graph g with two roots l and r. These roots determine the left-hand side (the *pattern*) and the right-hand side of the rule. Variables in R are represented by so called *empty nodes*, i.e. nodes labeled with the special 0-ary symbol $\perp$. For rewrite rules we have the usual restrictions, e.g. the left-hand side should not be just a variable, and all variables occurring on the right-hand side should also occur on the left-hand side. For convenience we will write $R \mid l$, $R \mid r$ for $g \mid l$, $g \mid r$ respectively.

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 mapping from the pattern of R to a subgraph of g that preserves the node structure. 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*. If h is the result of rewriting redex Δ in g this will be denoted by $g \xrightarrow{\Delta} h$, or just $g \rightarrow h$.

A collection of graphs and a set of rewrite rules can be combined into a *(term) graph rewrite system* (TGRS). More formally, a TGRS is a tuple $\mathcal{S} = \langle \mathcal{G}, \mathcal{R}\rangle$ where $\mathcal{R}$ is a set of rewrite rules, and $\mathcal{G}$ is a set of $\perp$-free graphs which is closed under $\mathcal{R}$-reduction.

In the sequel we will used the following notational conventions. Let $\mathcal{S} = \langle \mathcal{G}, \mathcal{R}\rangle$ be a TGRS. $\Sigma_{\mathcal{S}}$ denotes symbols in Σ that appear in $\mathcal{G}$ or in $\mathcal{R}$. The set of *function symbols* of $\mathcal{S}$ are those symbols for which there exist a rule in $\mathcal{R}$; the others are called *data symbols*. We only consider so called *function-data systems*, in which the left-hand side of each rewrite rule constists of a function root and data symbols elsewhere.

Modelling higher order functions

Since in TGRS's symbols have a fixed arity, it is impossible to use functions as arguments or to deliver functions as a result. However, higher order functions can be modelled in TGRS's by associating to each symbol S with nonzero arity a collection of data symbols (so called *Curry variants* of S). An application of such a Curry variant represents a partial application of S. Furthermore, a special *apply rule* is added to the TGRS for adding new arguments to such partial applications. The notions of types and type assignment as introduced in the present paper can deal with this concept of higher order functions. For a more elebarate description of this method the reader is referred to the full version of this paper.

3. Typing graphs

In this section we will define a notion of simple type assignment to graphs using a type system based on traditional systems for functional languages. The approach is similar to the one introduced in Bakel *et al.* [1992]. It is meant to illustrate the concept of 'classical' typing for graphs. In section 5 a different typing system will be described. We assume that $\mathbb{V}$ is a set of *type variables*, and $\mathbb{C}$ a set of *type constructors* with *arity* in $\mathbb{N}$.

3.1. Definition. (i) The set $\mathbb{T}$ of (*graph*) *types* is defined inductively as follows.

$$\mathbb{T} = \mathbb{V} \mid \mathbb{T} \rightarrow \mathbb{T} \mid \mathbb{C}\,\vec{\mathbb{T}}$$

(ii) The set $\mathbb{T}_{\mathrm{S}}$ of *symbol types* is defined as

$$\sigma_1, \ldots, \sigma_k, \tau \in \mathbb{T} \quad \Rightarrow \quad (\sigma_1, \ldots, \sigma_k) \Rightarrow \tau \in \mathbb{T}_{\mathrm{S}}, \qquad k \geq 0.$$

We will usually abbreviate $() \Rightarrow \tau$ to τ and $(\sigma) \Rightarrow \tau$ to $\sigma \Rightarrow \tau$.

In the sequel, $\alpha, \beta, \alpha_1, \ldots$ range over type variables; $\sigma, \tau, \tau_1, \ldots$ range over (symbol) types. Notions as *substitution* and *instance* are defined as expected. We use the denotation $\sigma \subseteq \tau$ to indicate that σ is an instance of τ.

In the rest of this section we describe how types can be assigned to graphs given a fixed type assignment to the (function and data) symbols by a so called *environment*. We assume that every environment contains the following type assignments for the symbols $\bot$ and **Ap**.

$$\mathcal{E}(\bot) = \alpha, \quad \mathcal{E}(\mathbf{Ap}) = (\alpha \rightarrow \beta, \alpha) \Rightarrow \beta.$$

Environment types are regarded as *type schemes*, i.e. in any concrete application of a symbol, one uses an instance of its environment type.

We consider new (basic) types to be introduced by so-called *algebraic type definitions*. In these type definitions a (possibly infinite) set of data symbols called *constructor* symbols is associated with each new type. The general form of an algebraic type definition for T is

$$T\,\vec{\alpha} = C_1\,\vec{\sigma_1} \mid C_2\,\vec{\sigma_2} \mid \cdots$$

Here $\vec{\alpha} \in \mathbb{V}$, and $\vec{\sigma_i} \in \mathbb{T}$ such that the variables appearing in $\vec{\sigma_i}$ are contained in $\vec{\alpha}$. Moreover, we assume that each C_i is a fresh constructor symbol. E.g., the type of lists could be obtained as follows.

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

A system $\mathcal{A}$ of algebraic type definitions induces a canonical type environment $\mathcal{E}_{\mathcal{A}}$ for all constructors introduced by $\mathcal{A}$. E.g. for the symbol **Cons** of the previous example $\mathcal{E}_{\mathcal{A}}$ will contain the following type declaration.

$$\mathbf{Cons} \; : \; (\alpha, \mathrm{List}(\alpha)) \Rightarrow \mathrm{List}(\alpha)$$

From now on let $\mathcal{A}$ be some fixed set of algebraic type definitions. We assume that every type environment $\mathcal{E}$ is consistent with the induced environment $\mathcal{E}_{\mathcal{A}}$.

3.2. Definition. Let $g = \langle N, \mathit{symb}, \mathit{args} \rangle$ be a graph. a function $\mathcal{T} : N \rightarrow \mathbb{T}$ is an *$\mathcal{E}$-typing for g* if for each $n \in g$ one has

$$(\mathcal{T}(n_1), \ldots, \mathcal{T}(n_k)) \Rightarrow \mathcal{T}(n) \subseteq \mathcal{E}(\mathit{symb}(n)).$$

where $k = \mathrm{arity}(\mathit{symb}(n))$, and $n_i = \mathit{args}(n)_i$. In case g is rooted we write $\mathcal{E} \vdash g : \sigma$ if $\mathcal{T}(r_g) = \sigma$ for some $\mathcal{E}$-typing $\mathcal{T}$ for g.

3.3. Example. Below a typed graph and the relevant part of the corresponding environment are indicated.

$$\begin{tikzcd} & \mathbf{F} : \mathrm{Int} \arrow[d] & \\ & \mathbf{Cons} : \mathrm{List(Int)} \arrow[dl] \arrow[dr] & \\ \mathbf{3} : \mathrm{Int} & & \mathbf{Nil} : \mathrm{List(Int)} \end{tikzcd}$$

$$\boxed{\begin{array}{lcl} \mathbf{F} & : & \mathrm{List}(\beta) \Rightarrow \beta, \\ \mathbf{Cons} & : & (\alpha, \mathrm{List}(\alpha)) \Rightarrow \mathrm{List}(\alpha), \\ \mathbf{Nil} & : & \mathrm{List}(\alpha), \\ \mathbf{3} & : & \mathrm{Int}. \end{array}}$$

As in λ-calculus, type assignment has the principal type property, i.e. if a graph g is typable then there exists a most general typing for g.

3.4. Definition. Let g be a graph, and $\mathcal{T}$ an $\mathcal{E}$-typing for g. $\mathcal{T}$ is a *principal* $\mathcal{E}$-typing for g if for any $\mathcal{E}$-typing $\mathcal{T}'$ for g one has $\mathcal{T}' \subseteq \mathcal{T}$.

3.5. Theorem. *There exists a computable function pt such that for any $\mathcal{E}$-typable graph g, $pt(g)$ is a principal $\mathcal{E}$-typing. Moreover $pt(g) = \mathsf{fail}$ if g is not $\mathcal{E}$-typable.*

3.6. Definition. Let $\mathcal{S} = \langle \mathcal{G}, \mathcal{R} \rangle$. A rewrite rule $R \in \mathcal{R}$ is *$\mathcal{E}$-typable* if there exists a typing $\mathcal{T}$ for g_R that meets the following requirements.

(1) $\mathcal{T}(l) = \mathcal{T}(r)$.
(2) $(\mathcal{T}(l_1), \ldots, \mathcal{T}(l_k)) \Rightarrow \mathcal{T}(l) = \mathcal{E}(\mathit{symb}(l))$.
(3) $\mathcal{T} \upharpoonright (R \mid l) = pt(R \mid l)$

Condition (2) states that the left root node should be typed exactly with the type assigned to the root symbol by the environment. This contrasts the requirement for applicative occurrences of the function symbol. Furthermore, condition (3) ensures that if R is applicable an actual typing of the graph (restricted to the matching part) is an instance of the typing of R (restricted to $R \mid l$).

We say that $\mathcal{R}$ is *$\mathcal{E}$-typable* if each $R \in \mathcal{R}$ is. The preservance of typability under reduction (sometimes called the *subject reduction property*) is indicated by the following result.

3.7. THEOREM. *Let $\mathcal{S} = \langle \mathcal{G}, \mathcal{R} \rangle$. Suppose $\mathcal{R}$ is $\mathcal{E}$-typable. Then for any $g, h \in \mathcal{G}$*

$$\mathcal{E} \vdash g : \sigma,\ g \rightarrow h \quad \Rightarrow \quad \mathcal{E} \vdash h : \sigma.$$

4. Usage analysis

Analyzing 'uniqueness' of nodes in a graph involves a function-argument dependency analysis. A first approach to a classification of 'unique' access to nodes in a graph is to count the references to each node. In practice, however, a more refined analysis is often possible, if one takes a specific reduction strategy into account. The idea is that multiple references to a node are harmless if one knows that only one of them remains at the moment of evaluation. E.g. the standard evaluation of a conditional statement **If** c **Then** t **Else** e causes first the evaluation of the c part, and subsequently evaluation of either t or e, but not both. Hence, a single access to a node n in t combined with a single access to n in e would overall still result in a 'unique' access to n.

This is generalized to arbitrary symbols: we assume that for each $S \in \Sigma_{\mathcal{S}}$, say with arity ℓ, the set $\{1, \ldots, \ell\}$ is divided into $k+1$ disjoint 'argument classes'

$$P, A_1, \ldots, A_k.$$

The intended meaning is that arguments occurring in P (*primary arguments*) are evaluated before any other argument whereas $A_1, \ldots, A_k$ are 'alternative' groups of *secundary arguments*: during the actual evaluation, arguments belonging to different groups are never evaluated both. Data symbols are considered to have only primary arguments. A reference (n, i) in g is called *primary* if i belongs to the primary arguments of $symb_g(n)$; otherwise (n, i) is *secundary*.

Now suppose p, q are paths from m to n such that p and q are disjoint between start and end point. Destructive access to n via p can be considered harmful if the first reference of p is primary and q starts with a secondary, or p, q start with references in a common argument class. In the former case access through p is done before access through q (notation $p \lhd q$); in the latter access via p, q might take place in any order ($p \sim q$). If p is indeed 'dangerous', destructive access to n via p will be prevented by *labeling* some reference in a tail part of p (containing only data nodes) as 'read-only' ($\otimes$). Possible 'write' access is indicated by $\odot$. A *marking function* is a labeling $use : Ref_g \rightarrow \{\odot, \otimes\}$ such that any 'suspect path' contains a marked reference as mentioned above. The treatment of cycles is somewhat subtle; see the full version of this paper.

Note that this leaves some freedom as to which reference is labeled $\otimes$. Some standard solutions are: 'last reference' marking, labeling the last reference on each suspect path as $\otimes$. A rude reference count marking would place $\otimes$ at every reference pointing to a node with reference count > 1.

5. Uniqueness typing

In this section we will extend the notion of typing (see section 3) with uniqueness information. We therefore consider the set of types $\mathbb{T}$ and attach to each subtype of a type a so called *uniqueness attribute* which may be $\bullet$ (for 'unique') or $\times$ (for 'ordinary' or 'nonunique').

The sets $\mathbb{U}$ and $\mathbb{U}^-$ of *uniqueness types* and *pre-uniqueness types* respectively, are defined by simultaneous induction, as follows.

$$\begin{array}{rcl} \mathbb{U} & = & \overset{\bullet}{\mathbb{U}^-} \,|\, \overset{\times}{\mathbb{U}^-}, \\ \mathbb{U}^- & = & \mathbb{V} \,|\, \mathbb{U} \to \mathbb{U} \,|\, \mathbb{C}\vec{\mathbb{U}}. \end{array}$$

For each $\sigma \in \mathbb{U}$ the *attribute* of σ is denoted by $[\sigma]$; e.g. $[\overset{\times}{\alpha}] = \times$ and $[\overset{\times}{\alpha} \overset{\bullet}{\to} \overset{\bullet}{\mathrm{Int}}] = \bullet$. The set $\mathbb{U}_\mathrm{S}$ of *uniqueness symbol types* is defined by

$$\sigma_1, \ldots, \sigma_k, \tau \in \mathbb{U} \;\Rightarrow\; (\sigma_1, \ldots, \sigma_k) \Rightarrow \tau \in \mathbb{U}_\mathrm{S}.$$

The notions of *substitution* and *instance* are defined straightforwardly. Moreover $|\sigma|$ denotes the (conventional) type obtained by removing uniqueness attributes.

The notion of type assignment is adapted in the following way. The correctness of a uniqueness-type assignment to a node in a graph now depends not only on the environment type for the symbol at that node and the types assigned to the argument nodes, but also on the way these arguments are accessed. Therefore the correctness of type assignment becomes dependent on the *global* structure of the graph, in contrast to the situation in conventional typing (relating types of nodes to types of their direct arguments).

The dependency on 'the way arguments are accessed' is made explicit below. We introduce so called *coercion relations* for access via $\odot$-references and $\otimes$-references respectively. The idea is that unique objects keep their uniqueness while being passed via $\odot$-arcs, and lose it when they are accessed via $\otimes$-references. Furthermore, a unique object can be coerced to a nonunique one (e.g. if such a nonunique object is required in a function application). Some care is needed in case of function types; see the full version of this paper.

Intuitively, a coercion statement $\sigma \leq \tau$ means that every σ-object can be regarded as a τ-object. For function types this view leads to the rule

$$\sigma' \leq \sigma, \tau \leq \tau' \;\Rightarrow\; \sigma \to \tau \leq \sigma' \to \tau'.$$

Note the so called contravariance in the first argument of the type constructor $\to$: one says that the first argument occurs on a *negative* position. This is generalized to arbitrary constructors: the coercion rules are made dependent on a 'negative/positive' classification of constructor arguments. For a given algebraic type system $\mathcal{A}$ one can deduce such a sign classification by standard (fixedpoint) techniques.

5.1. Definition. (i) The *coercion relation* $\leq$ is defined in two steps. On the set of uniqueness attributes $\{\bullet, \times\}$ one has

$$\bullet \leq \bullet, \qquad \bullet \leq \times, \qquad \times \leq \times.$$

Moreover on $\mathbb{U}$

$$\begin{array}{rcl} & \overset{u}{\alpha} \leq \overset{u}{\alpha}, & \\ \sigma' \leq \sigma, \tau \leq \tau' & \Rightarrow & \sigma \overset{u}{\to} \tau \leq \sigma' \overset{u}{\to} \tau', \\ u \leq u',\ \vec{\sigma} \leq \vec{\sigma'} & \Rightarrow & \overset{u}{T}\vec{\sigma} \leq \overset{u'}{T}\vec{\sigma'}, \end{array}$$

where $\vec{\sigma} \leq \vec{\sigma'}$ indicates componentwise coercion according to the sign classification of T. Set $\leq^{\odot} = \leq$.

(ii) The restricted coercion relation $\leq^{\otimes}$ is defined by

$$\sigma \leq^{\otimes} \tau \ \Leftrightarrow\ \sigma \leq \tau \text{ and } [\tau] = \times.$$

The notion of *environment* is extended to uniqueness typing, with the modification that a uniqueness type environment may contain several uniqueness variants of the same conventional type. This is necessary, e.g., for algebraic constructors. More formally, a uniqueness type environment is a function $\mathcal{E} : \Sigma_{\mathcal{S}} \to \wp(\mathbb{U}_{\mathrm{S}})$ such that

$$\mathcal{E}(\bot) = \{\overset{\bullet}{\alpha}, \overset{\times}{\alpha}\}, \quad \mathcal{E}(\mathbf{Ap}) = \{(\overset{s}{\alpha} \overset{t}{\to} \overset{u}{\beta}, \overset{s}{\alpha}) \Rightarrow \overset{u}{\beta} \mid s, t, u \in \{\,\bullet\,, \times\}\}.$$

The constructor type associated with an algebraic type system $\mathcal{A}$ can be supplied with uniqueness information. One could consider each version obtained by consistently attributing the variables and constructor symbols in an algebraic definition. For lists this would result in four uniqueness variants for **Cons**:

Cons	:	$(\overset{\times}{\alpha}, \overset{\times}{\mathrm{List}}(\overset{\times}{\alpha})) \Rightarrow \overset{\times}{\mathrm{List}}(\overset{\times}{\alpha})$	(1)	**Cons**	:	$(\overset{\bullet}{\alpha}, \overset{\times}{\mathrm{List}}(\overset{\bullet}{\alpha})) \Rightarrow \overset{\times}{\mathrm{List}}(\overset{\bullet}{\alpha})$	(3)
Cons	:	$(\overset{\times}{\alpha}, \overset{\bullet}{\mathrm{List}}(\overset{\times}{\alpha})) \Rightarrow \overset{\bullet}{\mathrm{List}}(\overset{\times}{\alpha})$	(2)	**Cons**	:	$(\overset{\bullet}{\alpha}, \overset{\bullet}{\mathrm{List}}(\overset{\bullet}{\alpha})) \Rightarrow \overset{\bullet}{\mathrm{List}}(\overset{\bullet}{\alpha})$	(4)

With (1) ordinary lists can be built. (2) can be used for lists of which the 'spine' is unique, (3) for lists containing unique elements, and (4) for lists of which both the spine and the elements are unique.

For consistency of uniqueness typing of rewrite rules it is necessary that uniqueness of pattern nodes is 'propagated upwards'. Therefore it makes sense to allow only *uniqueness propagating* constructor types: the resulting type τ of a constructor type $\mathbf{C} : \vec{\sigma} \Rightarrow \tau$ should be unique whenever one of its argument types $\vec{\sigma}$ is unique, thus rejecting (3) above; see the full paper.

5.2. Definition. Let $g = \langle N, \mathit{symb}, \mathit{args}\rangle$ be a graph.

(i) A function $\mathcal{U} : N \to \mathbb{U}$ is an *$\mathcal{E}$-uniqueness typing for g* if there is a marking function *use* for g such that for each $n \in g$ there exists $\sigma \in \mathcal{E}(\mathit{symb}(n))$ and $\tau_1, \ldots, \tau_k \in \mathbb{U}$ with

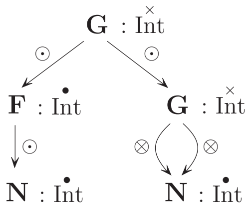
$$\begin{array}{c} \mathcal{U}(n_i) \leq^{u_i} \tau_i, \\ (\tau_1, \ldots, \tau_k) \Rightarrow \mathcal{U}(n) \subseteq \sigma, \end{array}$$

where $n_i = \mathit{args}(n)_i$, and $u_i = \mathit{use}(n, i)$ for $i \leq k = \mathrm{arity}(\mathit{symb}(n))$. The denotation $\mathcal{E} \vdash g : \sigma$ extends to this typing system.

(ii) $\mathcal{U}$ is a *plain uniqueness typing* if for each n as above

$$(\mathcal{U}(n_1), \ldots, \mathcal{U}(n_k)) \Rightarrow \mathcal{U}(n) \subseteq \sigma.$$

5.3. Example. The following gives (parts of) a well-typed graph and the corresponding environment.



$$\boxed{\begin{array}{lcl} \mathbf{F} & : & \overset{\bullet}{\alpha} \Rightarrow \overset{\bullet}{\alpha}, \\ \mathbf{G} & : & (\overset{\times}{\beta}, \overset{\times}{\beta}) \Rightarrow \overset{\times}{\beta}, \\ \mathbf{N} & : & \overset{\bullet}{\mathrm{Int}} \end{array}}$$

5.4. Definition. Let $\mathcal{S} = \langle \mathcal{G}, \mathcal{R} \rangle$. A rule $R \in \mathcal{R}$ is *$\mathcal{E}$-uniqueness typable* if there is a function $\mathcal{U} : g_R \to \mathbb{U}$ such that the following conditions are satisfied.

(1) $|\mathcal{U}|$ is a typing for R,
(2) $\mathcal{U}$ is a plain uniqueness typing for $R \mid l$,
(3) $\mathcal{U}$ is a uniqueness typing for $R \mid r$,
(4) $\mathcal{U}(r) \leq^u \mathcal{U}(l)$, where u is $\otimes$ if r is on a cycle, otherwise $\odot$.
(5) $\mathcal{E}(symb(l)) = \{(\mathcal{U}(l_1), \ldots, \mathcal{U}(l_k) \Rightarrow \mathcal{U}(l)\}$.

Some explanations are in place. The left-hand side of a rule is to be typed plainly, to ensure that the typing of the rule applies to any well-typed matching part of a concrete graph. The coercion condition for r is included to account for the effect of redirection in the construction of the contractum.

It can be proved that uniqueness typing is preserved under reduction, provided that the rewrite rules respect the argument classification of function symbols w.r.t. the way pattern nodes are passed from the left-hand to the right-hand side. The proof is complex and can be found in the full paper. The difficult part is the construction of a new marking of the result graph.

5.5. Theorem. *Let $\mathcal{S} = \langle \mathcal{G}, \mathcal{R} \rangle$. Suppose $\mathcal{R}$ is $\mathcal{E}$-uniqueness typable. Then for any $g, h \in \mathcal{G}$*

$$\mathcal{E} \vdash g : \sigma, \; g \to h \quad \Rightarrow \quad \mathcal{E} \vdash h : \sigma.$$

6. Locality properties

The intention of uniqueness typing is that uniqueness assumptions made in rewrite rules translate into locality properties of the actual arguments of the function involved. We call a node n in g *local* for m if $m \in p$ for all $p : r_g \rightsquigarrow n$. Of course such a property can only be expected if the evaluation order follows the argument classification.

A redex $\Delta = \langle R, \mu \rangle$ is called *primary* if $\mu(l)$ is reachable from r_g via a path consisting of primary references only.

6.1. Locality Theorem. *Let g be a uniqueness-typable graph. Let $\Delta = \langle R, \mu \rangle$ be a primary redex in g with uniqueness typing $\mathcal{U}$. Then for all $n \in R \mid l$*

$$[\mathcal{U}(n)] = \bullet \quad \Rightarrow \quad \mu(n) \textit{ is local for } \mu(l) \textit{ in } g.$$

7. Applications

In this section we will give some examples that show how the two problems mentioned in the introduction can be solved by using uniqueness types.

Consider the following list reversing function which can be implemented as a ‘destructive’ function if the given uniqueness type is used.

$$
\boxed{
\begin{array}{lllll}
\mathbf{Rev} : \overset{\bullet}{\text{List}}(\overset{\times}{\alpha}) \Rightarrow \overset{\bullet}{\text{List}}(\overset{\times}{\alpha}) & & \mathbf{Rev}\ (l) & \rightarrow & \mathbf{H}(l, \mathbf{Nil}) \\
\mathbf{H} : (\overset{\bullet}{\text{List}}(\overset{\times}{\alpha}), \overset{\bullet}{\text{List}}(\overset{\times}{\alpha})) \Rightarrow \overset{\bullet}{\text{List}}(\overset{\times}{\alpha}) & & \mathbf{H}\ (\mathbf{Nil}, r) & \rightarrow & r \\
 & & \mathbf{H}\ (\mathbf{Cons}(h,t), r) & \rightarrow & \mathbf{H}(t, \mathbf{Cons}(h, r))
\end{array}
}
$$

Note that not only the space of the obsolete **Cons** can be re-used, but also parts of its contents. In fact it already suffices to redirect the reference to t such that it points to r.

The second example shows how a predefined function **WriteChar** can be typed, having as input an object of type File and a character that should be written to the given file. The output consists of the modified file which is also unique, so it can be used for further writing.

$$
\boxed{\mathbf{WriteChar} : (\overset{\bullet}{\text{File}}, \overset{\times}{\text{Char}}) \Rightarrow \overset{\bullet}{\text{File}}}
$$

A more elaborated example, presenting an efficient quicksort algorithm that performs the sorting *in situ* on the data structure, can be found in Smetsers *et al.* [1993]. This algorithm is based on the same idea as used in the list reversal example.

References

Achten P.M., J.H.G. van Groningen and M.J. Plasmeijer, High Level Specification of I/O in Functional Languages, in: *Proc. of International Workshop on Functional Languages*, Glasgow, UK, Springer Verlag, 1993.

Bakel S. van, S. Smetsers and S. Brock, Partial Type Assignment in Left-Linear Term Rewriting Systems, in: J.C. Raoult, editor, *Proc. of 17th Colloqium on Trees and Algebra in Programming (CAAP'92)*, pages 300–322, Rennes, France, Springer Verlag, LNCS 581, 1992.

Barendregt H.P., M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer and M.R. Sleep, Towards an Intermediate Language based on Graph Rewriting, in: *Proc. of Parallel Architectures and Languages Europe (PARLE)*, pages 159–175, Eindhoven, The Netherlands, Springer Verlag, LNCS 259 II, 1987.

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

Barendsen Erik and Sjaak Smetsers, Graph Rewriting and Copying, Technical Report 92-20, University of Nijmegen, 1992.

Guzmán J.C. and P. Hudak, Single-Threaded Polymorphic Lambda Calculus, in: *Proc. of 5th IEEE Symp. on Logic in Computer Science*, pages 333–343, Philadelphia, PA, IEEE Computer Society Press, 1990.

Jacobs B.P.F., Conventional and Linear Formulas in a Logic of Coalgebras, Typescript, University of Utrecht, 1993.

Nöcker E.G.J.M.H., J.E.W. Smetsers, M.C.J.D. van Eekelen and M.J. Plasmeijer, Concurrent Clean, in: *Proc. of Parallel Architectures and Languages Europe (PARLE'91)*, pages 202–219, Eindhoven, The Netherlands, Springer Verlag, LNCS 505, 1991.

Smetsers Sjaak, Erik Barendsen, Marko van Eekelen and Rinus Plasmeijer, Guaranteeing Safe Destructive Updates through a Type System with Uniqueness Information for Graphs, Technical report, University of Nijmegen, 1993.

Wadler P., Linear types can change the world!, in: *Proc. of Working Conference on Programming Concepts and Methods*, pages 385–407, Israel, North Holland, 1990.

Wadsworth C.P., *Semantics and Pragmatics of the Lambda Calculus*, PhD thesis, Oxford University, 1971.