# Guaranteeing Safe Destructive Updates through a Type System with Uniqueness Information for Graphs

Sjaak Smetsers, Erik Barendsen, Marko van Eekelen, Rinus Plasmeijer

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

**Abstract.** In this paper we present a type system for graph rewrite systems: *uniqueness typing*. It employs usage information to deduce whether an object is 'unique' at a certain moment, i.e. is only locally accessible. In a type of a function it can be specified that the function requires a unique argument object. The correctness of type assignment guarantees that no external access on the original object will take place in the future. The presented type system is proven to be correct. We illustrate the power of the system by defining an elegant quicksort algorithm that performs the sorting *in situ* on the data structure.

## 1 Introduction

Some operations on complex data structures (such as arrays) cannot be implemented efficiently without allowing a form of destructive updating. For convenience, we speak about those functions as 'destructively using' their arguments. In case of graph-like implementations of functional languages without any precautions, this destructive usage is dangerous: on the level of the underlying model of computation this appears when arguments are shared between two functions.

However, in some specific cases destructive updates are safe, e.g. when it is known that access on the original object is not necessary in the future. We call such an object (locally) 'unique'.

Sharing/update analysis is used to find spots where destructive updates are possible. However, some functions require that a destructive update can be done in all contexts in which the function is applied. Such updating functions are functions for file I/O, array manipulation, interfacing with existing FORTRAN or C libraries, window-based I/O and functions that require an efficient storage management (e.g. in situ sorting of a large data structure). This requirement can be explicitly specified via a type system. This paper presents a type system related to linear types: uniqueness types. The uniqueness type system is defined for graph rewrite systems. It employs usage information to deduce whether the uniqueness attribute can be assigned to a type for a subgraph. A type which has the uniqueness attribute is also called a unique type. For functions that require an object of unique type, the type system guarantees that no external access on the original object will be possible anymore. So, (depending on the use of the

object in the function body) this information can be used to destructively update the unique object. A compiler can exploit uniqueness types by generating code that automatically updates unique arguments when possible. This has important consequences for the time and space behaviour of functional programs. The type system has been implemented for 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]).

The structure of the paper is as follows: first graph rewrite systems are briefly introduced using standard terminology (Section 2). Then, a notion of typing is defined for graph rewrite systems in Section 3. Section 4 describes a use analysis that provides important information that is necessary to assign uniqueness attributes. How uniqueness attributes are assigned is defined in Section 5. The extension to algebraic type definitions is given in Section 6. The correctness of the type system is proven in Section 7. Section 8 illustrates how reasoning about programs with uniqueness types can be done, after which Section 9 discusses related work.

## 2 Graph rewriting

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

### Graphs

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 $\Sigma$ is a (possibly infinite) set of *symbols* with *arity* in $\mathbb{N}$.

**Definition 1.** (i) A *labeled graph* (over $\langle \mathcal{N}, \Sigma \rangle$) is a triple

$$g = \langle N, symb, args \rangle$$

such that

(a) $N \subseteq \mathcal{N}$; $N$ is the set of *nodes* of $g$;
(b) $symb : N \to \Sigma$; $symb(n)$ is the *symbol* at node $n$;
(c) $args : N \to N^*$ such that $\text{length}(args(n)) = \text{arity}(symb(n))$.

Thus $args(n)$ specifies the outgoing edges of $n$. The $i$-th component of $args(n)$ is denoted by $args(n)_i$.

(ii) A *rooted graph* is a quadruple

$$g = \langle N, symb, args, r \rangle$$

such that $\langle N, symb, args \rangle$ is a labeled graph, and $r \in N$. The node $r$ is called the *root* of the graph $g$.

(iii) The collection of all finite rooted labeled graphs over $\langle \mathcal{N}, \Sigma \rangle$ is indicated by $\mathbb{G}$.

*Convention.* (i) $m, n, n', \ldots$ range over nodes; $g, g', h, \ldots$ range over (rooted) graphs.

(ii) If $g$ is a (rooted) graph, then its components are referred to as $N_g$, $symb_g$, $args_g$ (and $r_g$) respectively.

(iii) To simplify notation we usually write $n \in g$ instead of $n \in N_g$.

**Definition 2.** (i) A *path* in a graph $g$ is a sequence $p = (n_0, i_0, n_1, i_1, \ldots, n_\ell)$ where $n_0, n_1, \ldots, n_\ell \in g$ are nodes, and $i_0, i_1, \ldots, i_{\ell-1} \in \mathbb{N}$ are 'edge specifications' such that $n_{k+1} = args(n_k)_{i_k}$ for all $k < \ell$. In this case $p$ is said to be a *path from* $n_0$ *to* $n_\ell$ (notation $p : n_0 \rightsquigarrow n_\ell$).

(ii) Let $m, n \in g$. $m$ is *reachable from* $n$ (notation $n \rightsquigarrow m$) if $p : n \rightsquigarrow m$ for some path $p$ in $g$.

**Definition 3.** Let $g$ be a graph and $n \in g$. The *subgraph of* $g$ *at* $n$ (notation $g \mid n$) is the rooted graph $\langle N, symb, args, n \rangle$ where $N = \{m \in g \mid n \rightsquigarrow m\}$, and $symb$ and $args$ are the restrictions (to $N$) of $symb_g$ and $args_g$ respectively.

## Graph rewriting

This section introduces some notation connected with graph rewriting. For a complete operational description the reader is referred to the papers mentioned earlier.

Rewrite rules specify transformations of graphs. Each rewrite rule is represented by a special graph containing two roots. These roots determine the left-hand side (the *pattern*) and the right-hand side of the rule. Variables are represented by special 'empty nodes'. 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* $\mu$ is a mapping from the pattern of $R$ to a subgraph of $g$ that preserves the node structure. The combination of a rule and a match 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 $\mu$. 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*.

**Definition 4.** Let $\perp$ be a special symbol in $\Sigma$ with arity 0. Let $g$ be a graph.

(i) The set of *empty nodes* of $g$ (notation $g^\circ$) is the collection

$$g^\circ = \{n \in g \mid symb_g(n) = \perp\}.$$

(ii) The set of *non-empty nodes* (or *interior*) of g is denoted by $g^\bullet$. So $N_g = g^\circ \cup g^\bullet$.

(iii) $g$ is *closed* if $g^\circ = \emptyset$.

The objects on which computations are performed are closed graphs; the others are used as auxiliary objects, e.g. for defining graph rewrite rules.

**Definition 5.** (i) A *term graph rewrite rule* (or *rule* for short) is a triple $R = \langle g, l, r\rangle$ where $g$ is a (possibly open) graph, and $l, r \in g$ (called the *left root* and *right root* of $R$), such that

(a) $(g \mid l)^\bullet \neq \emptyset$;

(b) $(g \mid r)^\circ \subseteq (g \mid l)^\circ$.

(ii) If $symb_g(l) = \mathbf{F}$ then $R$ is said to be a *rule for* $\mathbf{F}$.

(iii) $R$ is *left-linear* if $g \mid l$ is a tree.

Here condition (1) expresses that the left-hand side of the rewrite rule should not be just a variable. Moreover condition (2) states that all variables occurring on the right-hand side of the rule should also occur on the left-hand side.

*Notation.* We will write $R \mid l$, $R \mid r$ for $g_R \mid l_R$, $g_R \mid r_R$ respectively.

**Definition 6.** Let $p, g$ be graphs. A *match* is a function $\mu : N_p \to N_g$ such that for all $n \in p^\bullet$

$$symb_g(\mu(n)) = symb_p(n),$$
$$args_g(\mu(n))_i = \mu(args_p(n)_i).$$

In this case we write $\mu : p \xrightarrow[m]{} g$.

**Definition 7.** Let $g$ be a graph, and $\mathcal{R}$ a set of rewrite rules.

(i) An $\mathcal{R}$-*redex* in $g$ (or just *redex*) is a tuple $\Delta = \langle R, \mu\rangle$ where $R \in \mathcal{R}$, and $\mu : (R \mid l) \xrightarrow[m]{} g$.

(ii) If $g'$ is the result of rewriting redex $\Delta$ in $g$ this will be denoted by $g \xrightarrow[\mathcal{R}]{\Delta} g'$, or just $g \xrightarrow[\mathcal{R}]{} g'$.

(iii) Let $\Delta = \langle R, \mu\rangle$ be a redex. The *redex root* of $\Delta$ (notation $\mathrm{r}(\Delta)$) is defined by

$$\mathrm{r}(\Delta) = \mu(r_R) \text{ if } r_R \in R \mid l,$$
$$= r_R \quad \text{otherwise.}$$

### Term graph rewrite systems

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

**Definition 8.** (i) A *term graph rewrite system* (TGRS) is a tuple $\mathcal{S} = \langle \mathcal{G}, \mathcal{R}\rangle$ 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.

(ii) $\mathcal{S}$ is *left-linear* if each $R \in \mathcal{R}$ is left-linear.
(iii) $\mathcal{S}$ is *regular* if for each $g \in \mathcal{G}$ the $\mathcal{R}$-redexes in $g$ are pairwise disjoint.
(iv) $\mathcal{S}$ is *orthogonal* if $\mathcal{S}$ is both left-linear and regular.

It can be shown that for a large class of orthogonal TGRSs (the so-called *interference-free* systems) the Church-Rosser property holds (see Barendsen & Smetsers [1992]).

*Notation.* Let $\mathcal{S} = \langle \mathcal{G}, \mathcal{R} \rangle$ be a TGRS. $\Sigma_{\mathcal{S}}$ denotes symbols in $\Sigma$ that appear in $\mathcal{G}$ or in $\mathcal{R}$. The set of *function symbols* of $\mathcal{S}$ (notation $\Sigma_{\mathcal{F}}$) are those symbols for which there exist a rule in $\mathcal{R}$. Moreover, $\Sigma_{\mathcal{D}} = \Sigma_{\mathcal{S}} \backslash \Sigma_{\mathcal{F}}$ denotes the set of *data symbols* of $\mathcal{S}$.

## 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 the next section a different typing system will be described.

**Definition 9.** Let $\mathbb{V}$ be a set of *type variables*, and $\mathbb{C}$ a set of *type constructors* with *arity* in $\mathbb{N}$. Write $\mathbb{C} = \mathbb{C}^0 \cup \mathbb{C}^1 \cup \ldots$ such that each $S \in \mathbb{C}^i$ has arity $i$.

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

$$\begin{aligned} \alpha \in \mathbb{V} &\Rightarrow \alpha \in \mathbb{T}, \\ C \in \mathbb{C}^k, \sigma_1, \ldots, \sigma_k \in \mathbb{T} &\Rightarrow C(\sigma_1, \ldots, \sigma_k) \in \mathbb{T}, \\ \sigma, \tau \in \mathbb{T} &\Rightarrow \sigma \to \tau \in \mathbb{T}. \end{aligned}$$

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

$$\begin{aligned} \sigma \in \mathbb{T} &\Rightarrow \sigma \in \mathbb{T}_S, \\ \sigma_1, \ldots, \sigma_k, \tau \in \mathbb{T} &\Rightarrow (\sigma_1, \ldots, \sigma_k) \to \tau \in \mathbb{T}_S. \end{aligned}$$

The *arity* of a symbol type is 0 if it is introduced by the first rule. Otherwise, the arity is $k$.

*Convention.* In the sequel, $\alpha, \beta, \alpha_1, \ldots$ range over type variables; $\sigma, \tau, \tau_1, \ldots$ range over (function) types.

**Definition 10.** (i) A *substitution* is a function $* : \mathbb{V} \to \mathbb{T}$.

(ii) Let $*$ be a substitution, and $\sigma \in \mathbb{T}_S$. The result of applying $*$ to $\sigma$ (notation $\sigma^*$) is inductively defined as follows.

$$\begin{aligned} \alpha^* &= *(\alpha), \\ (C(\sigma_1, \ldots, \sigma_k))^* &= C(\sigma_1^*, \ldots, \sigma_k^*), \\ (\sigma \to \tau)^* &= \sigma^* \to \tau^*, \\ ((\sigma_1, \ldots, \sigma_k) \to \tau)^* &= (\sigma_1^*, \ldots, \sigma_k^*) \to \tau^*. \end{aligned}$$

(iii) $\sigma$ is an *instance* of $\tau$ (notation $\sigma \subseteq \tau$) if there exists a substitution $*$ such that $\tau^* = \sigma$.

(iv) $\sigma$ and $\tau$ are *isomorphic* if $\tau^{*_1} = \sigma$ and $\sigma^{*_2} = \tau$ for some substitutions $*_1, *_2$. We will usually identify isomorphic types, i.e. types that result from each other by consistent renaming of type variables. That is, we regard types as type *schemes*.

## Applicative graph rewrite systems

In TGRS's symbols have a fixed arity. Consequently, it is impossible to use functions as arguments or to yield functions as a result. However, higher order functions can be *modeled* in TGRS's by associating to each symbol $\mathbf{S}$ with arity($\mathbf{S}$) $\geq 1$ a 0-ary constructor $\mathbf{S_0}$, and by adding a special *apply rule* (with function symbol $\mathbf{Ap}$) to the TGRS for supplying these new constructors with arguments.

For example, Combinatory Logic (CL) expressed by

$$\begin{aligned} \mathbf{S}\,xyz &\to xz(yz) \\ \mathbf{K}\,xy &\to x \\ \mathbf{I}\,x &\to x \end{aligned}$$

can be modeled in the following TGRS (using a self-explanatory linear notation).

$$\begin{aligned} \mathbf{S}(x,y,z) &\to \mathbf{Ap}(\mathbf{Ap}(x,z),\mathbf{Ap}(y,z)) \\ \mathbf{K}(x,y) &\to x \\ \mathbf{I}(x) &\to x \\ \mathbf{Ap}(\mathbf{Ap}(\mathbf{Ap}(\mathbf{S_0},x),y),z) &\to \mathbf{S}(x,y,z) \\ \mathbf{Ap}(\mathbf{Ap}(\mathbf{K_0},x),y) &\to \mathbf{K}(x,y) \\ \mathbf{Ap}(\mathbf{I_0},x) &\to \mathbf{I}(x) \end{aligned}$$

Note that each new constructor symbol introduces a new rule for $\mathbf{Ap}$.

**Definition 11.** Let $\mathcal{S} = \langle \mathcal{G}, \mathcal{R} \rangle$ be a TGRS.

(i) Let $\mathbf{S} \in \Sigma_{\mathcal{S}}$ with arity $\geq 1$. The above symbol $\mathbf{S_0} \in \Sigma_{\mathcal{D}}$ is called the *Curry variant* of $\mathbf{S}$.

(ii) The set $\Sigma_{\mathcal{C}} \subseteq \Sigma_{\mathcal{D}}$ denotes the set of Curry variants of $\Sigma_{\mathcal{D}}$ with arity $\geq 1$.

(iii) We say that $\mathcal{S}$ is *Curry complete* if $\mathcal{R}$ contains an $\mathbf{Ap}$-rule for each symbol $\mathbf{S}$ with arity $\geq 1$, as described above, and no other $\mathbf{Ap}$-rules.

(iv) Let $R \in \mathcal{R}$. The *principal node* of $R$ (notation p($R$) is $l_R$ if $symb(l_R) \neq \mathbf{Ap}$; otherwise it is the node containing $\mathbf{S_0}$.

*Assumption.* From now on we assume that all TGRS's are Curry complete.

### Assigning types to symbols

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*.

Currying imposes a restriction on type environments, that is to say, the type of a Curry variant $\mathbf{S_0}$ should be related to the type assigned to $\mathbf{S}$. We also assume a standard type for the symbol $\mathbf{Ap}$ to be declared.

**Definition 12.** (i) Let $\sigma = (\sigma_1, \ldots, \sigma_k) \to \tau$ be a function type. The *curried version* of $\sigma$ (notation $\sigma^{\mathcal{C}}$) is

$$\sigma^{\mathcal{C}} = \sigma_1 \to (\sigma_2 \to (\cdots(\sigma_k \to \tau)\cdots)).$$

(ii) A (*type*) *environment* for $\mathcal{S}$ is a function $\mathcal{E} : \Sigma_{\mathcal{S}} \to \mathbb{T}$ such that
(a) $\mathcal{E}(\bot) = \alpha$,
(b) $\mathcal{E}(\mathbf{Ap}) = (\alpha \to \beta, \alpha) \to \beta$,
(c) $\mathcal{E}(\mathbf{S_0}) = (\mathcal{E}(\mathbf{S}))^{\mathcal{C}}$.

### Algebraic data types

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

The general form of an algebraic type definition for $T$ is

$$\begin{aligned} T\,\boldsymbol{\alpha} &= C_1\,\boldsymbol{\sigma_1} \\ &= C_2\,\boldsymbol{\sigma_2} \\ &= \ldots \end{aligned}$$

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

$$\begin{aligned} \mathrm{List}(\alpha) &= \mathbf{Cons}(\alpha, \mathrm{List}(\alpha)) \\ &= \mathrm{Nil} \end{aligned}$$

A set $\mathcal{A}$ of algebraic type definitions induces a type environment $\mathcal{E}_{\mathcal{A}}$ for all constructors introduced by $\mathcal{A}$. More specifically, Let $C_i$ be the $i^{th}$ constructor defined by some algebraic type $T$. The $\mathcal{E}_{\mathcal{A}}$ type of $C_i$ is

$$\mathcal{E}_{\mathcal{A}}(C_i) = \boldsymbol{\sigma_i} \to T\,\boldsymbol{\alpha}.$$

*Convention.* Let $\mathcal{A}$ be a set of type definitions. $\Sigma_{\mathcal{A}}$ denotes the constructor symbols that are defined via some definition in $\mathcal{A}$.

*Assumption.* In the sequel we will assume that all constructors in $\mathcal{S}$ that are not the curried variant of some other symbol, are introduced by an algebraic type definition (i.e. $\Sigma_{\mathcal{D}} \backslash \Sigma_{\mathcal{C}} \subseteq \Sigma_{\mathcal{A}}$.)

## Assigning types to graphs

**Definition 13.** Let $g = \langle N, symb, args \rangle$ be a graph.

(i) A *type assignment* to $g$ (or *g-typing*) is a function $\mathcal{T} : N \to \mathbb{T}$.

(ii) Let $\mathcal{T}$ be a $g$-typing, and $n \in g$. The *function type* of $n$ according to $\mathcal{T}$ (notation $\mathcal{F}_{\mathcal{T}}(n)$) is defined as

$$\mathcal{F}_{\mathcal{T}}(n) = (\mathcal{T}(n_1), \ldots, \mathcal{T}(n_l)) \to \mathcal{T}(n)$$

where $l = \text{arity}(symb(n))$, and $n_i = args(n)_i$.

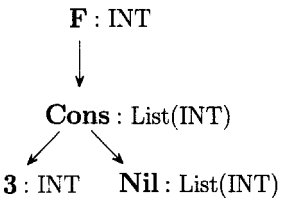
(iii) Let $\mathcal{E}$ be an environment. $\mathcal{T}$ is a *g-typing according to* $\mathcal{E}$ if for each $n \in g$ there exists a substitution $*$ such that

$$\mathcal{F}_{\mathcal{T}}(n) = \mathcal{E}(symb(n))^*.$$

*Example 1.* Let $\mathcal{E}$ be an environment containing the following type declarations.

$$\begin{array}{ll} \mathbf{F} & : \text{List}(\beta) \to \beta, \\ \mathbf{Cons} & : (\alpha, \text{List}(\alpha)) \to \text{List}(\alpha), \\ \mathbf{Nil} & : \text{List}(\alpha), \\ \mathbf{3} & : \text{INT}. \end{array}$$

Below, a graph and its typing according to $\mathcal{E}$ are indicated.




**Definition 14.** Let $\mathcal{S} = \langle \mathcal{G}, \mathcal{R} \rangle$ be a TGRS, and $\mathcal{A}$ a set of algebraic type definitions. Furthermore, let $\mathcal{E}$ be an environment for $\mathcal{S}$ .

(i) $R \in \mathcal{R}$ is *typable according to* $\mathcal{E}$ if there exist an $g_R$-typing $\mathcal{T}$ (according to $\mathcal{E}$) that meets the following requirements.

(a) $\mathcal{T}(l) = \mathcal{T}(r)$.

(b) $\mathcal{F}_{\mathcal{T}}(\text{p}(R)) = \mathcal{E}(symb(\text{p}(R)))$.

(ii) $\mathcal{R}$ is *typable* if there exists an environment $\mathcal{E}$ extending $\mathcal{E}_{\mathcal{A}}$ such that each $R \in \mathcal{R}$ is typable according to $\mathcal{E}$.

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.

Notice that the latter condition also provides that the abovementioned way of typing rewrite rules is essentially the same as the Mycroft type assignment system for the lambda calculus, see Mycroft [1981]. A Milner-like type assignment system (see Milner [1978]) can be obtained by stating this condition for *all* occurrences of a symbol **F** in the rule for **F**.

It is possible to formulate conditions under which typing is preserved during reduction; cf. Bakel *et al.* [1992]. We will not go into this here.

## 4 Usage 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. This can be achieved by taking into account the evaluation order dictated by a specific reduction strategy. This requires a nonlocal analysis of dependency of nodes in a graph. 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

$$\textbf{If } c \textbf{ Then } t \textbf{ Else } e$$

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 $n$. It is important to note that this property only holds if execution proceeds according to the chosen strategy; it may be disturbed if one allows reduction of arbitrary redexes.

Firstly, we introduce a classification principle for function arguments. The usage analysis presented in the rest of this section will be parametric in the chosen classification.

*Assumption.* Let $\mathcal{S}$ be a TGRS.

(i) Let $\mathbf{F} \in \Sigma_{\mathcal{F}}$, say with arity $l$. Assume that $\{1, \ldots, l\}$ is divided into $k+1$ disjoint 'argument classes'

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

The intended meaning is that arguments occurring in $P$ are evaluated before any other argument (*primary arguments*) 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. Furthermore, references via primary arguments to the graph are released before the graph is accessed via one of the secundary arguments.

(ii) A symbol is called *simple* if it has only primary arguments.

(iii) Each data symbol is simple.

**Remark 15.** We assume that the argument classification is consistent with each reduction rule, i.e. the way the arguments of a left-hand side are passed to functions in the corresponding right-hand side does not conflict with the respective argument classifications.

We will now describe a sharing analysis based on the above argument classification. First the argument dependency of functions is translated into dependency relations on paths and references in graphs.

**Definition 16.** Let $g \in \mathbb{G}$.

(i) The set of *references* of $g$ (notation $Ref_g$) is defined by

$$Ref_g = \{(m,i) \mid m \in g, i \leq \text{arity}(symb_g(m))\}.$$

(ii) Let $n \in g$. The set of *accesses* of $n$ (notation $acc(n)$) is

$$acc(n) = \{(m,i) \in Ref_g \mid args_g(m)_i = n\}.$$

We will now define dependency relations $\sim$ and $\lhd$.

**Definition 17.** (i) For each symbol $\mathbf{S}$ as above, and $i,j \leq l$, write $i \sim_{\mathbf{S}} j$ if $i,j$ belong to the same argument class of $\mathbf{S}$. Moreover, $i \lhd_{\mathbf{S}} j$ if $i \in P$ and $j \notin P$. The relation $\lesssim_{\mathbf{S}}$ is the union of $\lhd_{\mathbf{S}}$ and $\sim_{\mathbf{S}}$.

(ii) Let $p$ be a nonempty path; say $(n,i)$ is its first reference. Then $i$ is called the *index* of $p$ (notation $[p]$).

(iii) Let $g \in \mathbb{G}$. The relation $\lesssim$ is defined on nonempty paths in $g$ starting with the same node $n$, by

$$p \lesssim q \;\Leftrightarrow\; [p] \lesssim_{symb_g(n)} [q].$$

**Definition 18.** Let $g \in \mathbb{G}$.

(i) Let $p = (n_0, i_0, \ldots, n_\ell)$ and $p' = (n'_0, i'_0, \ldots, n'_{\ell'})$ be paths. Then $p$ and $p'$ *join* (notation $p \wedge p'$) if $p, p'$ start in the same node and are distinct elsewhere. More precisely, $p \wedge p'$ if

$$n_0 = n'_0,$$
$$n_{i+1} \neq n'_{j+1} \text{ for all } i < \ell,\ j < \ell'.$$

Note that in particular for paths $p$ of length 0 one has $p \wedge p$.

(ii) Let $p$ be a path in $g$, and $a \in Ref_g$. Then $p$ is *extendible* with $a$ if $p : n \rightsquigarrow m$ and $a = (m,i)$ for some $n, m, i$. The *extension* of $p$ with $a$ (defined in the obvious way) is denoted by $p * a$.

(iii) By $p \underset{a,a'}{\wedge} p'$ we will denote that $p \wedge p'$ and $p$, $p'$ are extendible with $a$, $a'$ respectively.

**Definition 19.** The relation $\lesssim$ induces a relation on $Ref_g$, as follows.

$$a \lesssim a' \;\Leftrightarrow\; p * a \lesssim p' * a' \text{ for some acyclic } p, p' \text{ with } p \underset{a,a'}{\wedge} p'.$$

**Definition 20.** A *critical path combination* is a quadruple $p, a, p', a'$ such that $p \underset{a,a'}{\wedge} p'$, the paths $p, p'$ are acyclic, $a \neq a'$, and $\arg(a) = \arg(a')$.

Suppose $p \underset{a,a'}{\wedge} p'$ is a critical path combination. If $p * a \lesssim p' * a'$, then the reference $a$ to $\arg(a)$ might be used before $a'$ (in the $\lhd$ case) or $a, a'$ might be used in any order. The idea is now that $\arg(a)$ is not allowed to be used destructively via $p * a$. This will be indicated by a suitable *labeling* of references with *use attributes* $\odot$ (for 'write use allowed') or $\otimes$ ('read access only').

A simple approach would label the reference $a$ above with $\otimes$ (see example 2). However, the usage information considered here is only important for *function* applications, in particular for parts of the graph matching the left-hand side of a rewrite rule. Since we consider systems with patterns (containing data nodes) we can be more liberal: in the above case it is sufficient that $p * a$ contains a reference labeled $\otimes$ anywhere in its 'data tail', to be made explicit below. The typing system will be such that this suffices to prevent destructive use of $\arg(a)$ via the indicated path.

**Definition 21.** (i) The set of *use attributes* is $U = \{\odot, \otimes\}$.

(ii) Let $use : Ref_g \to U$ be a labeling. A path $p$ in $g$ is *marked* if there exist paths $p_1, p_2$ and a reference $a$ such that $p = p_1 * a * p_2$, $p_2$ is a data path, and $use(a) = \otimes$.

(iii) A labeling $use$ is a *marking* for $g$ if for each critical path combination $p \underset{a,a'}{\wedge} p'$ one has

$$p * a \lesssim p' * a' \;\Rightarrow\; p * a \text{ is marked.}$$

There are two important examples of marking functions.

*Example 2.* Let $g$ be a graph.

(i) 'Last reference' marking is done as follows. Define $use^{\ell}$ as follows. Let $n \in g$. Then for each $a \in acc_g(n)$

$$\begin{aligned} use^{\ell}(a) &= \otimes \quad \text{if } a \lesssim a' \text{ for some } a' \in acc_g(n) \text{ with } a' \neq a, \\ &= \odot \quad \text{otherwise.} \end{aligned}$$

Note that this definition completely specifies the function $use^{\ell}$.

(ii) Straightforward reference counting is done by considering each symbol as simple and performing last reference marking. More directly, this labeling is obtained by setting

$$\begin{aligned} use^{\mathrm{rc}}(a) &= \odot \text{ if } \arg(a) \text{ has reference count 1,} \\ &= \otimes \text{ otherwise.} \end{aligned}$$

Using the standard classification of arguments of the conditional **IF**, and no specific assumptions about other symbols, the above two examples give the following markings of the displayed graph.


Now we can formulate which redexes are allowed to be contracted, in terms of the use function.

**Definition 22.** (i) Let $g \in \mathbb{G}$, and $m, n \in g$. Then $m$ is *local for* $n$ (*in* $g$) if

$$\forall p : r_g \rightsquigarrow m \ \ [n \in p].$$

(ii) Let $\Delta = \langle R, \mu \rangle$ be a redex in $g$. We say that $\Delta$ is *applicable* if for all $i$

$$use_g(\mu(l))_i = \odot \ \Rightarrow \ args_g(\mu(l))_i \text{ is local for } \mu(l).$$

The intention is that at least the redexes chosen by the strategy are applicable.

# 5 Uniqueness typing

## Uniqueness types

The use analysis described so far only takes the reduction strategy into account; not the particular structure of the rewrite rules. The use attributes of arguments may change during reduction, e.g. the $\odot$ attribute of a certain argument may change into a $\otimes$ after is redex has been contracted.

However, for a function $F$ that destructively uses one of its arguments it should be guaranteed that at the moment $F$ is evaluated the argument has a $\odot$ attribute. One way to ensure this is to require that this property holds at the moment the application of $F$ is built and that is remains valid during reduction.

The aim of the rest of this paper is to present a 'type system' in which the above-mentioned analysis can be performed.

The fact that a function may use one or more of its arguments destructively is expressed in its 'uniqueness type'. The syntax of these types is given in the following definition.

**Definition 23.** (i) The set $\mathbb{U}$ of *uniqueness types* is defined inductively by

$$\begin{gathered} \bullet, \times \in \mathbb{U}, \\ u, v \in \mathbb{U} \ \Rightarrow \ u \xrightarrow{\times} v \in \mathbb{U}, \\ u \xrightarrow{\bullet} v \in \mathbb{U} \end{gathered}$$

(ii) The set $\mathbb{U}^\bullet$ of *unique types* is defined by

$$\mathbb{U}^\bullet = \{u \in \mathbb{U} \mid u = \bullet \text{ or } u = v \xrightarrow{\bullet} w \text{ for some } v, w \in \mathbb{U}\}.$$

Moreover, $\mathbb{U}^\times = \mathbb{U} \backslash \mathbb{U}^\bullet$.

(iii) The set $\mathbb{U}_S$ of *uniqueness symbol types* is defined as

$$\begin{gathered} u \in \mathbb{U} \ \Rightarrow \ u \in \mathbb{U}_S, \\ u_1, \ldots, u_k, v \in \mathbb{U} \ \Rightarrow \ (u_1, \ldots, u_k) \rightarrow v \in \mathbb{U}_S. \end{gathered}$$

The constants $\bullet$ and $\times$ represent 'unique use' and 'potentially multiple use' respectively. The arrows are annotated to distinguish unique function objects from unique objects without specified structure, and nonunique function objects from general nonunique objects. In the following example this will be illustrated.

*Example 3.* Suppose **Upd** denotes a binary function which destructively updates its first argument with its second argument. So, the intended $\mathbb{U}$-type of **Upd** is something of the form $(\bullet, \times) \to u$. It is natural to require that the uniqueness of the updated object is propagated. Thus one arrives at the following type for **Upd**.

$$\mathbf{Upd} : (\bullet, \times) \to \bullet$$

A partial application of **Upd** to some unique expression $g$ results in a function $\mathbf{Ap}(\mathbf{Upd_0}, g)$ that may not be copied. For, if copying would be allowed, then each of the applications of a copy of the function would be allowed to update the first argument $g$ destructively, as is illustrated by the expression $\mathbf{G}(\mathbf{Ap}(\mathbf{Upd_0}, g), h)$ assuming the rule

$$\mathbf{G}(f, x) \to \mathbf{Pair}(\mathbf{Ap}(f, x), \mathbf{Ap}(f, x)),$$

which is obviously unwanted.

In our type system the $\mathbb{U}$-type of the above expression $\mathbf{Ap}(\mathbf{Upd_0}, g)$ will be $\times \xrightarrow{\bullet} \bullet$ which will prevent it from being copied.

However, in any context in which a nonunique nonfunctional $\mathbb{U}$-type is expected it is harmless to offer a unique object. This gives rise to a subtype hierarchy specifying which types are convertible (can be *coerced*) to other types. These coercions are defined as an ordering on $\mathbb{U}$. They are not only depending on the demanded and offered types of the context but also on the way the offered object is accessed. If the use information of graphs is not taken into account, some graphs are wrongly accepted. For this reason we define a coercion relation that also depends on the use value of the reference via which the corresponding part of the graph is accessed.

**Definition 24.** The orderings $\leq^{\odot}$ and $\leq^{\otimes}$ on $\mathbb{U}$ are defined as follows.

(i) Coercions via $\odot$-references are generated by

$$\begin{gathered}
\bullet \leq^{\odot} \bullet, \\
\times \leq^{\odot} \times, \\
\bullet \leq^{\odot} \times, \\
u_1 \leq^{\odot} u_2, v_1 \leq^{\odot} v_2 \;\Rightarrow\; u_2 \xrightarrow{\bullet} v_1 \leq^{\odot} u_1 \xrightarrow{\bullet} v_2, \\
u_2 \xrightarrow{\times} v_1 \leq^{\odot} u_1 \xrightarrow{\times} v_2.
\end{gathered}$$

(ii) Coercions via $\otimes$-references are the following.

$$\begin{gathered}
\times \leq^{\otimes} \times, \\
\bullet \leq^{\otimes} \times, \\
u_1 \leq^{\odot} u_2, v_1 \leq^{\odot} v_2 \;\Rightarrow\; u_2 \xrightarrow{\times} v_1 \leq^{\otimes} u_1 \xrightarrow{\times} v_2.
\end{gathered}$$

Since we do not have type variables the notion of type instance has to be adjusted slightly. Intuitively, a type $u$ is an instance of a type $v$ if $u$ has 'more structure' than $v$. This is made precise in the following definition.

**Definition 25.** The relation $\subseteq$ on $\mathbb{U}$ is defined as:

$$\begin{aligned} \bullet \subseteq \bullet, \quad & u \xrightarrow{\bullet} v \subseteq \bullet, \\ \times \subseteq \times, \quad & u \xrightarrow{\times} v \subseteq \times, \\ u_1 \subseteq u_2, v_1 \subseteq v_2 \Rightarrow \; & u_1 \xrightarrow{\bullet} v_1 \subseteq u_2 \xrightarrow{\bullet} v_2, \\ & u_1 \xrightarrow{\times} v_1 \subseteq u_2 \xrightarrow{\times} v_2. \end{aligned}$$

If $u \subseteq v$ we say that $u$ is an ($\mathbb{U}$-*type*) *instance* of $v$.

## Currying

As we have seen, in some cases it can be dangerous to copy references to functions. To prevent a 'dangerous' function from being copied it is distinguished from 'safe' functions by typing it with an arrow type supplied with a $\bullet$ attribute. The observation that once a symbol has been applied to a unique argument it may not be copied anymore (see example 3) leads to the following Currying rule.

**Definition 26.** (i) Let $u \in \mathbb{U}$. The *uniqueness attribute* of $u$ (notation $[u]$ is defined as follows.

$$\begin{aligned} [u] &= \bullet, \quad \text{if } u \in \mathbb{U}^\bullet \\ &= \times, \quad \text{if } u \notin \mathbb{U}^\bullet. \end{aligned}$$

(ii) For $\mathbf{u} = (u_1, \ldots, u_k)$ and $j \leq k$ the *cumulative uniqueness attribute up to* $j$ (notation $[\mathbf{u}]_j$) is defined by

$$\begin{aligned} [\mathbf{u}]_j &= \bullet \text{ if } [u_i] = \bullet \text{ for some } i \leq j, \\ &= \times \text{ otherwise.} \end{aligned}$$

(iii) Let $u = (u_1, \ldots, u_k) \to v$. The set of *curried versions* of $u$ (notation $u^{\mathcal{C}}$) is

$$\begin{aligned} u^{\mathcal{C}} = \{ & u_1 \xrightarrow{\times} (u_2 \xrightarrow{[\mathbf{u}]_1} \cdots (u_k \xrightarrow{[\mathbf{u}]_{k-1}} v) \cdots), \\ & u_1 \xrightarrow{\bullet} (u_2 \xrightarrow{[\mathbf{u}]_1} \cdots (u_k \xrightarrow{[\mathbf{u}]_{k-1}} v) \cdots) \}. \end{aligned}$$

The effect of applying a (possibly curried) function to a unique argument is that the result of the application itself becomes unique. One could say that uniqueness information 'propagates upwards'.

The correspondence between a symbol (with arity $\geq 1$) and its Curry variant is given by that **Ap** rule. In contrast to the (ordinary) type system presented in section 3, **Ap** can be used with different $\mathbb{U}$ which are *not* instances of one type. To make such 'generic' functions possible we allow the type environment to contain more than one type for each symbol.

**Definition 27.** An (*applicative*) *uniqueness type environment* is a function $\mathcal{E} : \Sigma \to \wp(\mathbb{U})$ such that

(1) $\mathcal{E}(\bot) = \{\bullet, \times\}$,

(2) $\mathcal{E}(\mathbf{Ap}) = \{(\times \xrightarrow{\times} \times, \times) \to \times, (\bullet \xrightarrow{\times} \times, \bullet) \to \times,$
$(\times \xrightarrow{\times} \bullet, \times) \to \bullet, (\bullet \xrightarrow{\times} \bullet, \bullet) \to \bullet,$
$(\times \xrightarrow{\bullet} \times, \times) \to \times, (\bullet \xrightarrow{\bullet} \times, \bullet) \to \times,$
$(\times \xrightarrow{\bullet} \bullet, \times) \to \bullet, (\bullet \xrightarrow{\bullet} \bullet, \bullet) \to \bullet\},$

(3) $\mathcal{E}(\mathbf{S_0}) \subseteq (\mathcal{E}(\mathbf{S}))^{\mathcal{C}}$.

Here $A^{\mathcal{C}} = \{a^{\mathcal{C}} \mid a \in A\}$.

### Assigning uniqueness types to graphs

Assigning $\mathbb{U}$-types to graphs can be done in two ways. The first way is comparable to standard type assignment (section 3). In the second way, the use attributes of the graph as well as coercions are taken into account.

**Definition 28.** Let $g = \langle N, symb, args\rangle$ be a graph, and $\mathcal{E}$ be an environment. Furthermore, let $\mathcal{U} : N \to \mathbb{U}$.

(i) Let $n \in g$. The *function type* of $n$ (notation $\mathcal{F}_{\mathcal{U}}(n)$) is

$$\mathcal{F}_{\mathcal{U}}(n) = (\mathcal{U}(n_1), \ldots, \mathcal{U}(n_l)) \to \mathcal{U}(n),$$

where $l = \text{arity}(symb(n))$, and $n_i = args(n)_i$.

(ii) $\mathcal{U}$ is an *uniqueness typing for $g$ according to $\mathcal{E}$* if for each $n \in g$ there exists $u \in \mathcal{E}(symb(n))$ such that

$$\mathcal{F}_{\mathcal{U}}(n) \subseteq u.$$

(iii) Let *use* be the function that supplies $g$ with use attributes. $\mathcal{U}$ is an *weighted uniqueness typing for $g$ according to $\mathcal{E}$* if for each $n \in g$ there exist $u \in \mathcal{E}(symb(n))$ and $v_1, \ldots, v_k \in \mathbb{U}$ such that

$$\mathcal{U}(n_i) \leq^{u_i} v_i,$$
$$(v_1, \ldots, v_k) \to \mathcal{U}(n) \subseteq u,$$

where $n_i = args(n)_i$, and $u_i = use(n)_i$ for $i \leq k = \text{arity}(symb(n))$.

(iv) If $\mathcal{U}$ is a (weighted) uniqueness typing for $g$, then the *type of $g$* (notation $\mathcal{U}(g)$) is simply $\mathcal{U}(r_g)$.

**Definition 29.** Let $\mathcal{S} = \langle \mathcal{G}, \mathcal{R}\rangle$ be a TGRS, and $\mathcal{A}$ a set of algebraic type definitions. Furthermore, let $\mathcal{E}$ be an environment.

(i) $R \in \mathcal{R}$ is *uniqueness-typable* (according to $\mathcal{E}$) if for each $u \in \mathcal{E}(symb(l))$ there exist a function $\mathcal{U} : g_R \to \mathbb{U}$ such that

(a) $\mathcal{U}$ is a uniqueness typing for $R \mid l$,
(b) $\mathcal{U}$ is a weighted uniqueness typing for $R \mid r$,
(c) $\mathcal{U}(r) \leq^{\odot} \mathcal{U}(l)$,
(d) $\mathcal{F}_{\mathcal{U}}(\text{p}(R)) = u$.

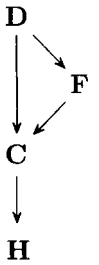
Such an $\mathcal{U}$ is called a *uniqueness typing* for $R$.

(ii) $\mathcal{R}$ is *uniqueness-typable* if there exists an environment $\mathcal{E}$ extending $\mathcal{E}_{\mathcal{A}}$, such that each $R \in \mathcal{R}$ is uniqueness-typable according to $\mathcal{E}$.

(iii) $\mathcal{S}$ is *uniqueness-typable* if there exists an uniqueness type environment $\mathcal{E}$ extending $\mathcal{E}_{\mathcal{A}}$ such that each $R \in \mathcal{R}$ as well as each $g \in \mathcal{G}$ is uniqueness-typable according to $\mathcal{E}$.

## 6 Algebraic type definitions

Since one allows pattern matching in function definitions, it is sometimes wrongly concluded that part of a pattern is unique. This appears e.g. in the following example, taking $\bullet \to \times$ for the constructor **C** and $\times \to \bullet$ for **F** with rule $\mathbf{F}(\mathbf{C}(x)) \to x$.




For this reason we require that (data) symbols appearing in a pattern of a rewrite rule also obey an 'upward propagation' rule, that is to say, if such a symbol expects one or more unique arguments the application itself is unique. E.g. in the above example **C** should be typed with $\bullet \to \bullet$, rejecting the given **F**-type.

Since the only symbols appearing in function patterns are constructors introduced by some algebraic type definition, the upward propagation requirement is obtained by making following assumption.

*Assumption.* Let $C \in \Sigma_{\mathcal{D}}$ with uniqueness type $(u_1, \ldots, u_k) \to v$. Then

$$u_i \in \mathbb{U}^\bullet \text{ for some } i \leq k \;\Rightarrow\; v \in \mathbb{U}^\bullet .$$

Consequently, a data object can only contain unique subparts if the object itself is unique. The fact that a symbol may have more than one environment type is also very useful for constructors. Remember, for example, the following algebraic type definition for lists.

$$\begin{aligned} \mathrm{List}(\alpha) &= \mathbf{Cons}(\alpha, \mathrm{List}(\alpha)) \\ &= \mathrm{Nil} \end{aligned}$$

A list of which the 'spine' is unique can be obtained by typing **Cons** by

$$\mathbf{Cons} : (\times, \bullet) \to \bullet .$$

A list with unique elements can be specified by assuming

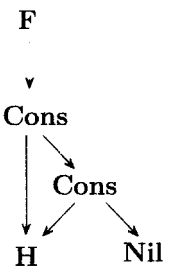
$$\mathbf{Cons} : (\bullet, \bullet) \to \bullet .$$

Notice that, because of the propagation rule, the uniqueness of elements implies the uniqueness of the spine.

Allowing both types for **Cons** simultaneously in the present type system may cause type conflicts. E.g. in the rule

$$\mathbf{F}(\mathbf{Cons}(x, y)) \to x,$$

**F** can be typed with $\bullet \to \bullet$. This is wrong, as is illustrated by the following application of **F**.




One way to solve this problem is to distinguish the different types of the constructors by introducing uniqueness *type constructors*. We only give an example.

*Example 4.* In the extended system, **Cons** can be typed as follows.

$$\mathbf{Cons} : (\bullet, \dot{\mathrm{List}}(\bullet)) \to \dot{\mathrm{List}}(\bullet),$$
$$\mathbf{Cons} : (\times, \dot{\mathrm{List}}(\times)) \to \dot{\mathrm{List}}(\times).$$

Then, a spine-unique list is typed with $\dot{\mathrm{List}}(\times)$ whereas the list containing also unique elements is typed with $\dot{\mathrm{List}}(\bullet)$.

This extension will not be elaborated here. However, to prevent incorrect type assignments we make the following assumption about type environments.

*Assumption.* If $\mathcal{E}$ is a uniqueness type environment, then the constructor types are chosen in such a way that the type conflicts mentioned above cannot occur.

## 7 Correctness

In order to show that uniqueness typing is preserved during reduction, some analysis with respect to the *use* function is needed. We focus on the relation between the uniqueness typing of a rewrite rule and the usage information of a graph before and after applying this rewrite rule. We will merely give an outline of the proof. The details will appear separately.

Fix an orthogonal TGRS $\mathcal{S} = \langle \mathcal{G}, \mathcal{R} \rangle$.

**Definition 30.** Let $\Delta = \langle R, \mu \rangle$ be a redex in $g$.

(i) Let $\mathcal{U} : R \to \mathbb{U}$. $\Delta$ is *$\mathcal{U}$-type correct* if $\mathcal{U}$ is a uniqueness typing for $R$ according to $\mathcal{E}$, and for each $n \in R \mid l$, $n \neq l$ (say $n = args(m)_i$) one has

$$\mathcal{U}(n) \in \mathbb{U}^\bullet \;\Rightarrow\; use_g(\mu(m))_i = \odot.$$

(ii) $\Delta$ is *type correct* if $\Delta$ is $\mathcal{U}$-type correct for some $\mathcal{U}$.

Note that the definition of 'applicable' (see 22) formulates a locality condition for the direct arguments of $\mu(l)$ only. The following result extends this property to all nodes in the matching fragment of the graph.

**Lemma 31.** *Let $\Delta$ be applicable and $\mathcal{U}$-type correct. Then for all $n \in (R \mid l) \cap (R \mid r)$ with $n \neq l$ one has*

$$\mathcal{U}(n) \in \mathbb{U}^\bullet \;\Rightarrow\; n \text{ is local for } \mu(l).$$

*Proof.* For 'ordinary' reduction rules, this follows from the propagation criterion for constructors and regularity of $\mathcal{S}$. For **Ap** reduction rules, the specific form of curry types and the predefined types for **Ap** imply the result. □

**Lemma 32.** *Let $m, n \in g$ with $(m, i) \in acc_g(n)$. Suppose $n$ is on a cycle not containing $m$. Then $use_g(m)_i = \otimes$.*

*Proof.* Examine the definition of *use*. □

**Proposition 33.** *Let $\Delta = \langle R, \mu \rangle$ be applicable in $g$. Say $g \xrightarrow[\pi]{\Delta} h$. Suppose $\Delta$ is $\mathcal{U}$-type correct, with $\mathcal{U}(r) \in \mathbb{U}^\bullet$. Then*

$$acc_h(\mathrm{r}(\Delta)) \subseteq acc_g(\mu(l)).$$

*Proof Sketch.* By the following case distinction.

*Case* 1. $\mathrm{r}(\Delta) \notin \mu(R \mid l)$. Then $\mathrm{r}(\Delta)$ is fresh in $h$, so $acc_h(\mathrm{r}(\Delta)) = acc_g(\mu(l))$ after redirection.

*Case* 2. $\mathrm{r}(\Delta) = \mu(n)$, $n \in \mu(R \mid l)$. Since $\mathcal{U}(n) \in \mathbb{U}^\bullet$ it follows by type correctness and lemma 31 that $\mu(n)$ is local for $\mu(l)$. Hence $\mu(l) \rightsquigarrow m$ for every $(m, i) \in acc_g(\mu(n))$. Now let $(m, i) \in acc_g(\mu(n))$. We want to show that $m$ is not present in $h$. If $m \in \mu(R \mid l)$ this is easily seen. Otherwise, if $m$ would be present in $h$ (after redirection and garbage collection), then $\mu(n) \rightsquigarrow m$ ($\rightsquigarrow \mu(n)$). Hence $use_g(\mu(m'))_i = \otimes$ for any $(m', i) \in acc_R(n)$, by lemma 32, contradicting type correctness of $\Delta$. Taking the effect of redirection into account it follows that $acc_h(\mu(n)) \subseteq acc_g(\mu(l))$. □

**Proposition 34.** *Let $\Delta$ be applicable and $\mathcal{U}$-type correct in $g$; say $g \xrightarrow[\pi]{\Delta} h$.*

(i) *Suppose $\mathcal{U}(r) \in \mathbb{U}^\bullet$. Then for all $(m, i) \in acc_g(\mu(l))$ such that $m$ is present in $h$ one has*

$$use_g(m)_i = \odot \;\Rightarrow\; use_h(m)_i = \odot.$$

(ii) *Let* $n \in R \mid r$ *with* $n \neq r$. *Suppose* $\mathcal{U}(n) \in \mathbb{U}^\bullet$. *Then for all* $(m,i) \in acc_R(n)$

$$use_R(m)_i = \odot \;\Rightarrow\; use_h(\hat{m})_i = \odot,$$

*where* $\hat{m}$ *denotes the* $h$*-node corresponding to* $m$.

*Proof Sketch.* (i) Suppose $use_g(m)_i = \odot$. By proposition 33 we only have to consider $acc_g(m)$ to determine $use_h(m)_i$. If $p \underset{m,m'}{\wedge} p'$ in $h$ causing $use_h(m)_i = \otimes$, then a redirection 'above' $\mu(l)$ has taken place. This can only occur if $\mu(l)$ is on a cycle in $g$, contradicting lemma 32.

(ii) By a case distinction, distinguishing the possible positions of $n, m$. Lemma 31 is used in the case $n \in R \mid l$ and $m \notin R \mid l$. □

**Proposition 35.** *Let* $\Delta$ *be applicable in* $g$; *say* $g \overset{\Delta}{\underset{\mathcal{R}}{\rightarrow}} h$. *Let* $n \in g$ *such that* $n \notin \mu(R \mid l)$, *and* $n \in h$. *Then for all* $(m,i) \in acc_g(n)$ *with* $m$ *present in* $h$ *one has*

$$use_g(m)_i = \odot \;\Rightarrow\; use_h(m)_i = \odot.$$

*Proof Sketch.* Suppose, towards a contradiction, $use_g(m)_i = \odot$ but $use_h(m)_i = \otimes$. Suppose this is caused by $m'$, i.e. $(m', i') \in acc_h(m)$ such that $m \sim m'$ or $m \lhd m'$, say $p \underset{m,m'}{\wedge} p'$ with $p \sim p'$ or $p \lhd p'$. Since this situation does not occur in $g$, these parts contain new nodes or new arcs. Distinguish two cases. If $\mathrm{r}(\Delta) \notin p, p'$ one arrives at a conflict with the argument classification (cf. remark 15). Assuming, on the other hand, $\mathrm{r}(\Delta) \in p$ or $\mathrm{r}(\Delta) \in p'$ leads to a contradiction with $use_g(m)_i = \odot$. □

For reduction on uniqueness-typed graphs, the above results imply a 'subject reduction' result: typing remains correct when reducing applicable redexes.

**Lemma 36.** *Let* $g \in \mathcal{G}$. *Suppose* $g$ *is uniqueness-typable. If* $\Delta$ *is applicable, then* $\Delta$ *is type correct.*

*Proof.* Obvious. □

**Lemma 37.** (i) *Let* $u, v, w \in \mathbb{U}$. *Then*

$$u \leq^{\odot} v,\; v \leq^{\otimes} w \;\Rightarrow\; u \leq^{\otimes} w.$$

(ii) *Let* $u, v, v' \in \mathbb{U}$. *Suppose* $u \leq^{\odot} v$ *and* $v' \subseteq v$. *Then there exists* $u' \in \mathbb{U}$ *with* $u' \subseteq u$ *and* $u' \leq^{\odot} v'$.

**Theorem 38.** *Suppose* $\mathcal{R}$ *is uniqueness-typable according to* $\mathcal{E}$. *Let* $\mathcal{U}$ *be a uniqueness typing for* $g$ *(according to* $\mathcal{E}$*). Furthermore, let* $g \overset{\Delta}{\underset{\mathcal{R}}{\rightarrow}} h$ *with* $\Delta$ *applicable. Then there exists a uniqueness typing* $\mathcal{U}'$ *for* $h$ *such that* $\mathcal{U}'(h) = \mathcal{U}(g)$.

*Proof.* $\mathcal{U}$ can be extended to a uniqueness typing of $h$ by defining it on the new nodes according to the type assignment to $\Delta$ (proposition 34 (ii)). The type assigned to the other nodes remains correct, as follows from propositions 34 (i, ii), 35 and lemma 37, by distinguishing the different kinds of nodes in $h$. □

# 8 Reasoning about programs with uniqueness types

Uniqueness types can be used in several contexts. When one wants to interface functional languages with imperative programs, one can assign a unique type to those arguments that are destructively updated by the imperative function. In this way file I/O and array updating can be incorporated without loosing the referential transparency. With these applications in mind it may seem that the destructive behaviour of the function has to be explicitly programmed using a non-functional programming language. However, it is of course also possible for a compiler to generate destructive updates for pure functions defined in the functional language itself. This is of great importance for improving the time-space behaviour of functional programs.

Below an example is given in a functional programming language of which it is assumed that uniqueness types are assigned on the underlying graph rewrite system (which can be derived directly from the program by removing some syntactical sugar). The language uses underlining to indicate that a type has the uniqueness attribute •. [ ] in a type denotes the List type. [ ] in a rule denotes the Nil element and [ a | b ] denotes Cons a b. ( ,..., ) denotes standard tupling. So, $\underline{[\ T\ ]}$ denotes a list of type T with a unique spine.

```
qs :: [ T ] → [ T ]
qs [ ]              =  [ ]
qs [ hd | tl ]      =  (qs left) ++ [ hd | qs right ]
                       where
                       (left, right) = split tl hd

split :: [ T ] → T → ([ T ],[ T ])
split [ ] p         =  ([ ], [ ])
split [ hd | tl ] p =  ([ hd | left ], right),   if p ≥ hd
                    =  (left, [ hd | right ])
                       where
                       (left, right) = split tl p
```

Compared with the imperative quick-sort algorithm the functionally written quick-sort algorithm qs has the disadvantage that the split function has to construct new lists for its result. Now, if the function split would be defined on a spine-unique list, the construction of the new cons nodes could be done by updating the old ones. Looking at the actual difference between the old cons node given as an argument to split ([hd | tl]) and the new cons node to be constructed (either [hd | left] or [hd | right]) it can be deduced that only the tail of the cons node has to be updated. This means that the split function does not create new cons nodes at all but is actually rearranging tail pointers in such a way that the ordered list is obtained. Such *in situ* updating is essential to be able to handle large data structures efficiently.

With respect to the updating the run-time behaviour of the functional program can be similar to its imperative counterpart. However, the specified program will require a relatively large recursion stack. Both split and qs can be

transformed to a tail recursive version using program transformations that also eliminate the construction of intermediate data structures. Tail recursion is usually translated into a loop on the machine code level. The applied transformation maintains the uniqueness of the types. So, for the resulting elegant functional program a compiler can generate code that is as efficient as the code for an imperatively written quick-sort algorithm. Hence, this example shows that uniqueness types solve one of the challenges set at the 1990 Dagstuhl seminar on functional languages (Johnsson [1990]).

```
qs :: [ T ] → [ T ] → [ T ]
qs [ ] tail                 = tail
qs [ hd | tl ] tail         = qs left [ hd | qs right tail ]
                              where
                              (left, right) = split tl hd [ ] [ ]

split :: [ T ] → T → [ T ] → [ T ] → ([ T ],[ T ])
split [ ] p left right      = (left, right)
split [ hd | tl ] p left right = split tl p [ hd | left ] right,  if p ≥ hd
                               = split tl p left [ hd | right ]
```

The reasoning about the programs above implicitly made certain assumptions about the generated code. It was assumed that updating was actually done whenever this was possible. More specifically, it was assumed that updates could actually take place for all objects of the same type. Using only such very general kinds of assumptions and the uniqueness type information the storage behaviour of the functional program was deduced and improved by a program transformation. It is important that these assumptions are further formalised. Any compiler should obey the resulting formal rules such that reasoning about the time and space behaviour of a functional program is independent of a specific compiler. The programmer then can deduce whether or not it is worthwhile to use uniqueness types for those cases where the efficiency of the time-space behaviour is critical. It seems that such reasoning is relatively simple and can be applied successfully to design time and space efficient purely functional programs for many kinds of real-life applications.

## 9 Related work

The update problem is also addressed (using linear types) in Wadler [1990] and Guzmán & Hudak [1991]. Both papers use lambda calculus as basic model hence requiring a more indirect kind of analysis. With the proposed approach in this paper graphs are used directly as the objects of consideration. The presented system for uniqueness types incorporates a solution to several of the questions raised in Wadler [1990]. Uniqueness types are in a sense orthogonal to the standard type systems for functional languages. The uniqueness type system has been used successfully to support high level I/O and efficient array handling. Experience with uniqueness types has shown an important change in the use of

functional languages from academic exercises to real-life programming (ranging from a window-based text editor to a relational database). The use function presented in Section 4 has been inspired by the analysis presented for *poly-lam*$_{st}$ in Guzmán & Hudak [1991] which is geared towards efficient array manipulation. They use Wadsworth's shared lambda calculus involving partly copying of lambda terms when functions are shared. In a certain sense the proposed uniqueness types are a generalisation of their single-threadedness analysis to a general graph rewriting context.

## 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 van, S, S. Smetsers and S. Brock, Partial Type Assignment in Left-Linear Term Rewriting Systems, in: *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, 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 Juan C. and Paul. Hudak, Single-Threaded Polymorphic Lambda Calculus, in: *Proc. of Logic in Computer Science (LICS'90)*, pages 333–345, Phildelphia, IEEE Computer Society Press, 1991.

Johnsson Thomas., Discussion Summary: which analysis?, in: *Proc. of Functional Languages: Optimization For Parallelism*, pages 4–5, Dagstuhl, Germany, Dagstuhl seminar, 1990.

Milner R.A., Theory of Type Polymorphism in Programming, *Journal of Computer and System Sciences*, 17, 1978.

Mycroft A., *Abstract interpretation and optimising transformations for applicative programs*, PhD thesis, University of Edinburgh, 1981.

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.