To appear in: proceedings of PLILP'95, LNCS.

Uniqueness Typing in Theory and Practice

Erik Barendsen Sjaak Smetsers
Computing Science Institute
University of Nijmegen*

Abstract

We present *uniqueness typing* as an extension of a familiar Curry style typing system for graph rewrite systems. This provides a method for combining purely functional programming with operations having side-effects, while preserving the functional nature of the language. We give a technical account as well as the underlying intuition.

The system is shown to be effective in the sense that a uniqueness variant of a given conventional type can be determined automatically. Finally, this paper gives some examples of the application of uniqueness typing in the programming language *Clean*.

1. Introduction

Traditional functional programming languages are unable to deal with operations with side effects. Indeed, by admitting these operations (such as file manipulations) one risks the loss of referential transparency since these involve changing ('destructively updating' or 'mutating') the state of an input object.

In recent years, various proposals have been brought up as solutions to this shortcoming. One of these is the restriction of destructive operations to input objects that occur only once. If formulated as a type system, this idea is related to *linear logic* (see Girard et al. (1989), Troelstra (1992)). In this logic, assumptions are regarded as *resources*. Some ('linear') resources can be used exactly once; others ('unlimited' resources) can be discarded, or used more than once. Wadler (1990) describes a *linear type system* for λ-calculus in which, e.g., files are considered as linear resources. Unlimited use is modelled by the 'resource duplication' operator ! from linear logic. (The logic without ! will be referred to as *pure* linear logic.)

Uniqueness typing is related to linear typing in that it also uses an occurrence counting mechanism. The type system, however, is more involved than linear typing. In fact, uniqueness typing is a combination of pure linear typing (yielding 'unique' objects) and traditional typing (for 'non-unique' objects without restrictions on their number of occurrences), connected by a subtyping mechanism. Non-unique objects should not be confused with the (duplicatewise unique) copies of linear resources produced by ! in linear logic.

As most implementations of functional languages use graph-like structures, we focus on graph rewriting as our model of computation. The *objects* in our typing system are *graphs* that are subject to rewrite rules in *term graph rewrite systems* (TGRS's, see Barendregt et al. (1987)). Multiple usage of a subobject corresponds to multiple *references* to a particular node. Our graphs may contain arbitrary sharing, including cyclic structures.

The *types* are annotated versions of the types in a traditional type system (referred to as 'conventional typing') which is present in most functional programming languages.

Conventional typing in TGRS's is defined in terms of local requirements for type assignment to nodes in a graph. In an application of a function **F**, say, these constraints relate the type of

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

the **F** node and the type of its argument nodes to the symbol type of **F**. The symbol types are declared in a so-called *type environment*. Note that the non-inductive nature of graphs prevents an more traditional (inductive) natural deduction formulation of typing.

Uniqueness typing formulates additional requirements on the reference structure of the graphs in question. In the types of functions, it can now be specified that a given argument should be unique (by the annotation $\bullet$). In any application, the concrete function argument should have reference count 1, so the function has indeed 'private' access to its argument. This is used for destructive operations: e.g., **WriteChar** : $(\mathsf{Char}^{\times}, \mathsf{File}^{\bullet}) \rightarrowtail \mathsf{File}^{\bullet}$. Here, $\times$ denotes 'non-unique'.

Sometimes, the uniqueness of unique objects is inessential, for example in the case of non-destructive functions like **ReadChar** : $\mathsf{File}^{\times} \rightarrowtail \mathsf{Char}^{\times}$, which can also be applied to a unique object. The technical tool to achieve this is a combination of reference counting and a subtyping relation.

This mechanism is still too restrictive for practical use, however. In many cases, depending on the evaluation strategy, one knows that a certain reference will be used (and disappear) before another reference to the same object.

Besides enabling the use of destructive updates, uniqueness typing can help to improve storage management: if it is clear from a rewrite rule that a certain function argument is discarded, and it is specified as unique, then it will certainly become obsolete after rewriting. Re-usage of its space can therefore be anticipated. This leads to so-called *compile-time garbage collection*.

The development of the uniqueness type system (introduced in Barendsen and Smetsers (1993a), see also (1993b)) has been guided by the following objectives.

First of all, it is natural to require that the type system is *consistent* in the sense that 'typed programs cannot go wrong'. This means that typability is preserved under evaluation, also known as the *subject reduction property*, which is proved to hold in the uniqueness type system in Barendsen and Smetsers (1993a).

Second, for practical use the type system should be *effective*: suitable types (if they exist) should be computable. For conventional typing this has been shown in Barendsen and Smetsers (1993a). The present paper deals with the decidability problem of the uniqueness type system.

A side condition (also mentioned by Guzmán and Hudak (1990)) was that the system must be *easy to use* and to reason about; expressing destructive functions should be natural. The presented type system fits to the well-known Hindley-Milner approach to type inference. A programmer writing traditional functional programs should not have to worry about uniqueness. Indeed, programs that are type correct in the Hindley-Milner sense remain so, by considering all types as non-unique. On the other hand, programmers should have maximal freedom to exploit uniqueness typing. There are no essential restrictions from a pragmatic point of view: arbitrary uniqueness variants of data structures and higher-order functions are allowed. For example, one can define functions taking destructive operations as arguments and/or deliver them as a result. It is even possible to introduce new data structures containing such destructive operations, cf. Achten et al. (1993).

The research presented here served as a basis for the incorporation of a uniqueness type system in the lazy functional language *Clean*. The most important application of uniqueness types so far is the implementation of an efficient high-level library for writing arbitrary interactive applications (see Achten et al. (1993) and Achten and Plasmeijer (1995)). Section 6 contains some examples written in *Clean* showing the merits of the type system.

Structure of this paper

We start with a brief introduction to graph rewriting. Section 3 summarizes earlier results on conventional typing. Section 4 contains an informal explanation of the uniqueness type system. This section is based on Barendsen and Smetsers (1993a). In Section 5, the system is extended with 'uniqueness polymorphism'. It gives a technical account, including a type derivation procedure. We give some examples in the programming language *Clean* in Section 6, and conclude with a discussion of related work (Section 7).

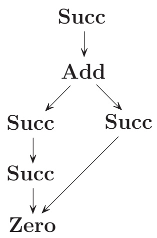
The reader who is interested in the global idea and the applications of uniqueness typing could, at first reading, skip Section 5.

2. Graph Rewriting

Term graph rewrite systems were introduced in Barendregt et al. (1987). This section summarizes some basic concepts which are provided by Barendsen and Smetsers (1992); see also (1994).

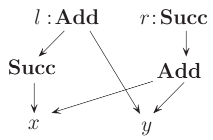
We will consider finite directed graphs in which each node is labelled with a *symbol*. Each symbol **S**, say, has a fixed arity determining the number of outgoing edges (the *arguments*) of any node labelled with **S**.

The following pictures shows a graph (with the topmost node considered as *root*) containing sharing.

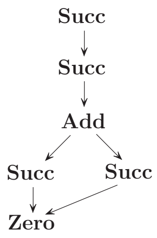


In its ultimate form, sharing results in cyclic structures.

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 (the *result*) of the rule. For example, the graph representation of the rule $\mathbf{Add}(\mathbf{Succ}(x), y) \to \mathbf{Succ}(\mathbf{Add}(x, y))$ is as follows.



A graph can be *rewritten* if one of the rules is applicable, i.e. the pattern of that rule *matches* the graph. If such a match has been determined, the matching part of the graph is replaced by (an instance of) the result of the rule involved. E.g. applying the previous rule to the example graph leads to the following result.



3. Type Assignment for Graphs

The work on *conventional typing*, reported in Barendsen and Smetsers (1993a) and (1993b), has been inspired by van Bakel et al. (1992) who consider TRS's. Our types are built up from type variables, the standard type constructor $\to$, and other type constructors (like List) that can be introduced via *algebraic type specifications*. An example of such a specification is

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

which specifies the type constructor List, and moreover the data constructors **Nil** and **Cons** (for building list objects and defining functions by pattern matching).

The symbols of graph rewrite systems are supplied with a type by a *type environment* $\mathcal{E}$. Such an environment contains declarations of the form

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

where k is the arity of $\mathbf{F}$. The part of $\mathcal{E}$ corresponding to the data constructors is determined by the algebraic specifications; e.g. for lists one has

$$\begin{array}{rcl} \mathbf{Nil} & : & \mathsf{List}(\alpha), \\ \mathbf{Cons} & : & (\alpha, \mathsf{List}(\alpha)) \rightarrowtail \mathsf{List}(\alpha). \end{array}$$

Due to the separation of specifications (rewrite rules, algebraic types) from applications one needs an *instantiation mechanism* to deal with different occurrences of symbols. E.g., for the identity function symbol $\mathbf{I}$ (with obvious rewrite rule), a type environment would normally contain $\mathbf{I} : \alpha \rightarrowtail \alpha$. This type is considered as schematic: at an applicative occurrence of $\mathbf{I}$ one can conclude $\mathbf{I} : \tau \rightarrowtail \tau$ for each τ.

The use of higher-order functions in TGRS's (in which each symbol has a fixed arity) is made possible by adding a *Currying* procedure. Our results can be applied to ordinary rewrite systems and purely applicative ones (with application as only function), as well as to hybrid variants (such as used in *Clean*). We will not go into this here.

The notion of typing cannot be defined inductively, but is specified in terms of local requirements for a type assignment to nodes.

Formally, a *typing* of a graph g is a function $\mathcal{T}$ assigning a type to each node of g, such that for any application

$$\begin{tikzcd}[column sep=small] & {} \arrow[d] & \\ & n : \mathbf{F} \arrow[dl] \arrow[dr] & \\ n_1 & \cdots & n_k \end{tikzcd}$$

say with $\mathcal{E}$ containing

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

there is an substitution $*$ such that

$$\mathcal{T}(n_1) = \sigma_1^*, \quad \ldots, \quad \mathcal{T}(n_k) = \sigma_k^*,$$
$$\mathcal{T}(n) = \tau^*.$$

For a fixed environment, each typable graph has a *principal type*, of which all other valid types are instances. The principal types can be computed effectively, which can be used to infer a suitable type environment for a given set of rewrite rules.

The typing concept of rewrite rules is based on principal typing. In Barendsen and Smetsers (1993a), the authors show that typing is consistent: typing is preserved during reduction. One can even avoid referring to principal types by imposing a syntactical restriction on rewrite systems. The resulting class of 'safe' rewrite systems is reasonably large and includes familiar programming languages such as *Miranda*.

4. Introduction to Uniqueness Typing

Uniqueness typing combines conventional typing and a reference count bookkeeping. All conventional types are now annotated with *uniqueness attributes*, which refer to the reference structure of graph objects. The attributes are $\bullet$ (for 'unique') and $\times$ (for 'non-unique').

We will informally describe the system in a few steps. A symbol type declaration

$$\mathbf{F} : (\sigma^\bullet, \tau^\times) \rightarrowtail \cdots$$

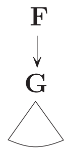
means that, apart from the conventional typing requirements, **F**-applications are restricted to those in which the first argument is unique, i.e. has reference count 1. There are no reference requirements for the second argument.

Also result types are attributed. Let us consider a function definition without pattern matching. Then there are three possibilities to produce a unique output: by returning one of the unique arguments, like in the projection

$$\mathbf{First} : (\sigma^\bullet, \tau^\times) \rightarrowtail \sigma^\bullet,$$

or by building a 'fresh' object by means of a data constructor. The third (which actually boils down to one of the previous possibilities) is applying a function with a unique result.

The effect of the result attribution is that an application

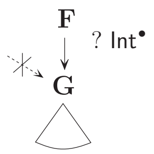


$$\begin{array}{|lcl|}\hline \mathbf{F} & : & \sigma^\bullet \rightarrowtail \cdots \\ \mathbf{G} & : & \cdots \rightarrowtail \sigma^\bullet \\ \hline \end{array}$$

can be considered correct in the displayed environment, and will remain so while **F**'s arguments is subject to reduction: the reference count of **F**'s argument will remain 1.

Types of symbols are again collected in a type environment. These types should be correct according to the rewrite rules for the function.

A simple type system would read as follows. Type correctness of applications is determined by the *requested* argument type (indicated below by $? \cdots$), the actual result type of the argument, and the reference count of that argument; see the example below.

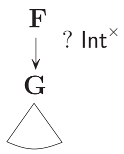


$$\begin{array}{|lcl|}\hline \mathbf{F} & : & \mathsf{Int}^\bullet \rightarrowtail \cdots \\ \mathbf{G} & : & \cdots \rightarrowtail \mathsf{Int}^\bullet \\ \hline \end{array}$$

In general, the requested type is an instance of **F**'s environment type (obtained by consistent substitution). The actual result type of **G** is determined by **G**'s environment type, in the same way.

Naturally, for non-unique objects there are no requirements on the reference structure of the application.

The above mechanism can be refined. One could imagine that an environment type for **G** indicates uniqueness of the result, which is not required by **F**'s environment type. In that case, uniqueness of **G**'s result is *discarded* when passed to **F**. The situation is then as follows.



$$\begin{array}{|lcl|}\hline \mathbf{F} & : & \mathsf{Int}^\times \rightarrowtail \cdots \\ \mathbf{G} & : & \cdots \rightarrowtail \mathsf{Int}^\bullet \\ \hline \end{array}$$

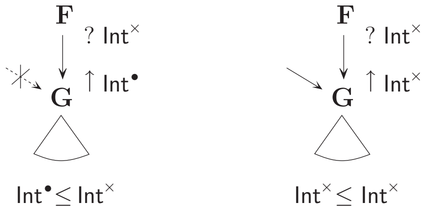
This is an acceptable application, if one regards $?\mathsf{Int}^\times$ as 'no reference requirements'. We express this by introducing a *subtyping relation* indicating e.g.

$$\mathsf{Int}^\bullet \leq \mathsf{Int}^\times.$$

Even if **G** has reference count larger that 1 this application would be harmless.

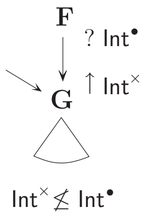
In short, the *offered type* (by an argument) is determined by the *result type* of its topmost symbol and the reference count of that argument. In the above application, **G**'s offered type (indicated below by $\uparrow \cdots$) is $\mathsf{Int}^\bullet$, because **G** has reference count 1.

The application is type correct if the argument's offered type (i.e. the type after correction) is a subtype of the requested type involved. The two cases mentioned above can now be visualised as follows.



$$\boxed{\begin{array}{lcl} \mathbf{F} & : & \mathsf{Int}^\times \rightarrowtail \cdots \\ \mathbf{G} & : & \cdots \rightarrowtail \mathsf{Int}^\bullet \end{array}}$$

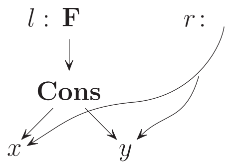
Correction of $\mathsf{Int}^\bullet$ to $\mathsf{Int}^\times$ expresses that the 'locally unique' result of **G** cannot be regarded as unique because of **G**'s reference count. Obviously, the following application is not type correct.



$$\boxed{\begin{array}{lcl} \mathbf{F} & : & \mathsf{Int}^\bullet \rightarrowtail \cdots \\ \mathbf{G} & : & \cdots \rightarrowtail \mathsf{Int}^\bullet \end{array}}$$

Having explained the general idea, we can now focus on some details.

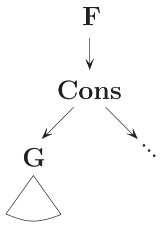
Because of the presence of data constructors in function definitions (pattern matching), the access of functions is not restricted to their direct arguments. For example, a function **F** defined like



has access to the arguments-of-its-argument after pattern matching. To ensure correctness of **F** applications, uniqueness assumptions on x, for instance, should be indicated in the type of **F**:

$$\mathbf{F} : \mathsf{List}(\mathsf{Int}^\bullet) \rightarrowtail \cdots.$$

Then an application of the form

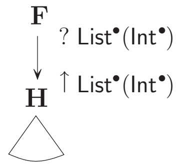


should be such that **G** is *unique* for **F**, i.e. has reference count 1, and moreover the **Cons**-node has reference count 1 (note that external access on the **Cons**-nodes can lead to external access on **G**, via pattern matching). In fact, the 'path reference count' of **G** should be 1: there should be exactly one path from **F** to **G**.

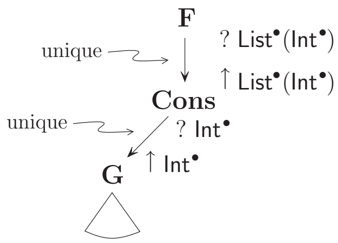
Since the pattern in the **F** application might be incomplete at the moment of typing, the 'path reference count' requirement needs to be split into referencewise requirements. This is achieved by a principle called *uniqueness propagation*: a uniqueness assumption on x should induce a uniqueness assumption on $\mathbf{F}$'s direct argument:

$$\mathbf{F} : \mathsf{List}^\bullet(\mathsf{Int}^\bullet) \rightarrowtail \cdots.$$

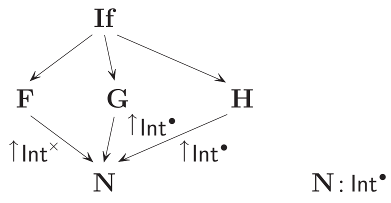
Uniqueness propagation is encoded in the types, for example by excluding the type $\mathsf{List}^\times(\mathsf{Int}^\bullet)$. As a consequence, any partially matching application



is required to have a unique first reference (i.e., there is no other reference to $\mathbf{H}$). The uniqueness of Int in $\mathbf{F}$'s argument type ensures that the second reference is also unique when the pattern has been completed by evaluating $\mathbf{H}$.



The second refinement concerns the way references are counted. Sometimes, multiple access to a unique argument is harmless if one knows that only one of the references will be present at the moment of evaluation. This leaves room for a, more sophisticated, referencewise determination of offered type $\uparrow \sigma$. This is explained in Barendsen and Smetsers (1993a). We illustrate this refined reference count by an example:



Indeed, if one evaluates the conditional

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

in the standard way, the condition is evaluated first (so access from $\mathbf{F}$ to $\mathbf{N}$ is not unique), and subsequently one of the alternatives is chosen and evaluated. Depending on this choice, either the reference of $\mathbf{G}$ to $\mathbf{N}$ or that of $\mathbf{H}$ to $\mathbf{N}$ is left and unique.

Our uniqueness typing system is parametric in the reference counting mechanism. We thus abstract from any particular (straightforward or refined) reference analysis: we assume that a classification of references into 'unique' and 'non-unique' is provided, indicated by a *marking* with labels $\odot$ and $\otimes$ (for unique and non-unique references respectively). The type correction leading to the offered type $\uparrow \cdots$ along a specific reference depends on the reference label. E.g. $\uparrow \mathsf{Int}^\bullet$ for $\odot$-references, $\uparrow \mathsf{Int}^\times$ for $\otimes$-references if the involved environment result type is $\mathsf{Int}^\bullet$. In case of $\mathsf{Int}^\times$, both offered versions are $\mathsf{Int}^\times$.

Barendsen and Smetsers (1993a) present an analysis of algebraic type definitions, leading to several uniqueness variants of the types of data constructors. For the list constructor **Cons**, for example, these are

$$\begin{array}{lll} \mathbf{Cons} & : & (\alpha^\times, \mathrm{List}^\times(\alpha^\times)) \rightarrowtail \mathrm{List}^\times(\alpha^\times) \quad (1) \\ \mathbf{Cons} & : & (\alpha^\times, \mathrm{List}^\bullet(\alpha^\times)) \rightarrowtail \mathrm{List}^\bullet(\alpha^\times) \quad (2) \\ \mathbf{Cons} & : & (\alpha^\bullet, \mathrm{List}^\bullet(\alpha^\bullet)) \rightarrowtail \mathrm{List}^\bullet(\alpha^\bullet) \quad (3) \end{array}$$

With (1) ordinary lists can be built. (2) can be used for lists of which the 'spine' is unique, and (3) for lists of which both the spine and the elements are unique. The variant containing $\mathrm{List}^\times(\alpha^\bullet)$ is rejected because of the abovementioned uniqueness propagation.

5. Uniqueness Typing

Formally, a uniqueness type is a pair $S = \langle\sigma, A\rangle$ consisting of a conventional type σ and an attribute function A, which assigns a uniqueness attribute to each node in the tree representation of σ. This syntax tree is particularly useful for formal definition of some concepts that we introduce below. We will occasionally refer to this representation, but usually denote the attributes as superscripts in the conventional type, for example $\mathsf{List}^\bullet(\mathsf{Int}^\times)$. By $|S|$ we denote the underlying conventional type σ.

The subtyping relation is defined using an auxiliary ordering $\leq$ on attributes, defined by $\bullet \leq \times$, $\bullet \leq \bullet$, $\times \leq \times$. We can use this relation also to indicate propagation requirements: an attribution $\mathsf{List}^u(\mathsf{Int}^v)$ is only correct if $u = \bullet$ whenever $v = \bullet$, or equivalently $u \leq v$. In general, propagation requirements for type constructors are determined by analyzing the algebraic specification of these, see Barendsen and Smetsers (1993a).

Attribute variables

To denote families of uniqueness types in a compact way, and to develop a notion of *most general attribution* we introduce uniqueness *schemes*. We extend the attribute set with *attribute variables* $a, b, \ldots$, and use *coercion statements* to indicate restrictions and relations between attribute variables.

For example, the type of the list constructor **Cons** is formulated as

$$\mathbf{Cons} : (\alpha^a, \mathsf{List}^b(\alpha^a)) \rightarrowtail \mathsf{List}^b(\alpha^a) \qquad [b \leq a].$$

In general, a symbol type declaration contains a *set* of coercion statements. An *instantiation* of a symbol type consists of an attribute assignment (mapping attribute variables to attributes) which does not conflict with the coercion statements involved, and an attributed instantiation of the conventional variables. For example, to obtain a spine-unique list one instantiates b with $\bullet$. Lists of unique integers are obtained by choosing $a := \bullet$, and subsequently $\alpha^\bullet := \mathsf{Int}^\bullet$. The coercion statement $b \leq a$ forces b to be instantiated with $\bullet$. In general, coercion statements are used to ensure that all instances are *internally consistent*: they respect the propagation properties of type constructors.

A uniqueness type assignment to a graph now consists of a conventional typing together with an attribution and a set of coercion statements. In order to express the validity of such typings we parameterize the subtyping relation $\leq$ on types with a set of coercion statements (a so-called *coercion environment*).

Coercion environments

Let Γ be a *coercion environment*, i.e. a finite set of coercion statements $u \leq v$, with u, v uniqueness attributes. We say that $u \leq v$ is *derivable* from Γ (notation $\Gamma \vdash u \leq v$) if $\Gamma \vdash u \leq v$ can be produced by the axioms

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

$$\Gamma \vdash u \leq u,$$
$$\Gamma \vdash u \leq \times, \qquad \Gamma \vdash \bullet \leq u$$

and rule

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

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

An attribute assignment $\diamond$ is said to *respect* Γ if $\Gamma^{\diamond}$ (defined in the obvious way) is consistent.

The relation $\leq$ is extended to types: the validity of $\langle \sigma, A \rangle \leq \langle \sigma, B \rangle$ in Γ depends subtypewise on the validity of $\Gamma \vdash A(p) \leq B(p)$ and/or $\Gamma \vdash B(p) \leq A(p)$ (with p a position in the syntax tree of σ). As usual, the directions of these positionwise inequalities depends on the covariance and contravariance of the type constructors. One has, for example,

$$b \leq a \vdash \mathsf{Int}^a \xrightarrow{\bullet} \mathsf{Bool}^c \leq \mathsf{Int}^b \xrightarrow{\bullet} \mathsf{Bool}^{\times}.$$

The attributes of corresponding occurrences of the type constructor $\rightarrow$ (in the left-hand and the right-hand side of an inequality) should be identical; see Barendsen and Smetsers (1993a) for a motivation. The same is required (to ensure substitutivity of the subtyping relation) for variables. E.g.,

$$\Gamma \vdash \mathsf{Int}^u \xrightarrow{v} \beta^w \leq \mathsf{Int}^{u'} \xrightarrow{v'} \beta^{w'} \quad \text{iff} \quad \Gamma \vdash u' \leq u, v = v', w = w'.$$

The relation between node type (of a function argument) and requested type (of a function), along $\odot$ and $\otimes$ references respectively, can be fully expressed in terms of coercion statements. This avoids an explicit type correction operation yielding the intermediate 'offered type' ($\uparrow \cdots$ in Section 4) in question. This is done by defining

$$\begin{array}{lcl} \Gamma \vdash S \leq^{\odot} T & \Leftrightarrow & \Gamma \vdash S \leq T, \\ \Gamma \vdash S \leq^{\otimes} T & \Leftrightarrow & \Gamma \vdash S \leq T, \ \times \leq \text{attribute}(T). \end{array}$$

Here attribute(T) denotes the topmost attribute of T.

Expressing consistency

For any type S, the set of coercion statements expressing internal consistency is denoted by $\mathrm{Con}(S)$. For example,

$$\mathrm{Con}(\mathsf{List}^u(\mathsf{Int}^v \xrightarrow{w} \mathsf{List}^y(\mathsf{Char}^z))) = [u \leq w, y \leq z].$$

Observe that the constructor $\rightarrow$ is propagating in neither of its arguments.

A *uniqueness type environment* consists of symbol type declarations of the form

$$F : (S_1, \ldots, S_k) \rightarrowtail T \qquad [\Gamma]$$

where Γ consists of attribute restrictions including internal consistency, i.e.

$$\Gamma \vdash \mathrm{Con}(S_1), \ldots, \mathrm{Con}(S_k), \mathrm{Con}(T).$$

For example (see also Section 6)

$$\mathbf{Append} : (\mathsf{List}^b(\alpha^a), \mathsf{List}^c(\alpha^a)) \rightarrowtail \mathsf{List}^d(\alpha^a) \qquad [b \leq a, c \leq a, d \leq a, c \leq d].$$

Uniqueness typing of graphs

Let $\mathcal{E}$ be a uniqueness type environment. An $\mathcal{E}$-typing for a graph g is a pair $\langle \mathcal{U}, \Gamma \rangle$, where $\mathcal{U}$ assigns a Γ-consistent uniqueness type to each node in g, such that for each application

$$\begin{tikzcd} & {} \arrow[d] & \\ & n:\mathbf{F} \arrow[dl, "m_1"'] \arrow[dr, "m_k"] & \\ n_1 & \cdots & n_k \end{tikzcd}$$

(where $m_1, \ldots, m_k$ are the markings of the references), say with $\mathcal{E}$ containing

$$\mathbf{F} : (S_1, \ldots, S_k) \rightarrowtail T \qquad [\Gamma_{\mathbf{F}}],$$

the following holds. There is an attribute assignment $\diamond$ respecting $\Gamma_{\mathbf{F}}$ and a consistent substitution $*$ on the resulting type $(S_1^\diamond, \ldots, S_k^\diamond) \rightarrowtail T^\diamond$ such that

$$\begin{gathered} \Gamma \vdash \Gamma_{\mathbf{F}}^\diamond, \\ \Gamma \vdash \mathcal{U}(n_1) \leq^{m_1} (S_1^\diamond)^*, \quad \ldots, \quad \Gamma \vdash \mathcal{U}(n_k) \leq^{m_k} (S_k^\diamond)^*, \\ \Gamma \vdash \mathcal{U}(n) = (T^\diamond)^*. \end{gathered}$$

It it easy to show that uniqueness typing is substitutive: if $\langle \mathcal{U}, \Gamma \rangle$ is a typing for g, and $\diamond$ is an attribute assignment respecting Γ, then $\langle \mathcal{U}^\diamond, \Gamma^\diamond \rangle$ is also a typing for g. This substitutivity also holds with respect to instantiation of conventional variables; see Barendsen and Smetsers (1993a).

Derivation of uniqueness types

Below, we will show that uniqueness types can be determined in an effective way. In fact, the typing algorithm described at the end of this section has been incorporated in the Nijmegen *Clean* compiler.

We proceed in two steps. First we show how to determine an attribute environment (if it exists) that completes a uniqueness type assignment to a uniqueness typing. Note that determining such an environment is indeed the essential step, since all restrictions on attributes can be expressed by coercion statements: a type $\mathsf{Int}^\bullet$ is equivalent to Int^a in any environment with $a \leq \bullet$.

Suppose we are given (in a fixed environment $\mathcal{E}$) a uniqueness type assignment $\mathcal{U}$ for g, such that its attribute-free version $|\mathcal{U}|$ is a conventional typing for g. Then we can decide whether $\mathcal{U}$ can be extended to a uniqueness typing for g. Moreover, if the answer is positive we can effectively calculate a suitable environment Γ containing the 'minimal requirements' on the attributes in $\mathcal{U}$.

The above $\Gamma = \Gamma^{\mathcal{E}}(g, \mathcal{U})$ is constructed by setting for each node n as above

$$\begin{aligned} \Gamma_n^{\mathcal{E}}(g, \mathcal{U}) \quad &= \quad \mathrm{Con}(\mathcal{U}(n)) \\ &\cup \quad \Gamma_{\mathbf{F}} \\ &\cup \quad \mathrm{Sub}^{m_1}(\mathcal{U}(n_1), S_1) \cup \cdots \cup \mathrm{Sub}^{m_k}(\mathcal{U}(n_k), S_k) \\ &\cup \quad \mathrm{Eq}(\mathcal{U}(n), T), \end{aligned}$$

where $\mathrm{Sub}^m(U, V)$ is the set of coercion statements expressing (for $m = \odot, \otimes$, and U, V with $|U|$ an instance of $|V|$) that U is $\leq^m$-related to an instance of V. This set depends on the attributes of U and V in the part of the syntax tree up to the variable nodes in V. Moreover $\mathrm{Eq}(U, V)$ (with the same requirements) contains subtypewise attribute equalities $u = v$. Finally we set

$$\Gamma^{\mathcal{E}}(g, \mathcal{U}) = \bigcup_{n \in g} \Gamma_n^{\mathcal{E}}(g, \mathcal{U}).$$

Now we have

$$\begin{array}{rcl} \Gamma^{\mathcal{E}}(g,\mathcal{U}) \text{ is consistent} & \Rightarrow & \langle \mathcal{U}, \Gamma^{\mathcal{E}}(g,\mathcal{U})\rangle \text{is an } \mathcal{E}\text{-uniqueness typing for } g; \\ \Gamma^{\mathcal{E}}(g,\mathcal{U}) \text{ is inconsistent} & \Rightarrow & \text{there is no } \Gamma \text{ such that } \langle \mathcal{U}, \Gamma\rangle \text{ is an} \\ & & \mathcal{E}\text{-uniqueness typing for } g. \end{array}$$

One can show that $\Gamma^{\mathcal{E}}(g,\mathcal{U})$ gives the minimal requirements for the attribution, i.e. if $\Gamma^{\mathcal{E}}(g,\mathcal{U})$ is consistent, then for any Γ' one has

$$\langle \mathcal{U}, \Gamma'\rangle \text{ is an } \mathcal{E}\text{-uniqueness typing for } g \;\Rightarrow\; \Gamma' \vdash \Gamma^{\mathcal{E}}(g,\mathcal{U}).$$

This environment is obviously computable from g, $\mathcal{U}$ and $\mathcal{E}$. The next step towards a typing algorithm is the verification that consistency is a decidable property of attribute environments. Given an attribute environment Γ, we can first solve inequalities involving the attribute constants $\bullet$ and $\times$. The result is a reduced environment and an attribute assignment. These are constructed from the original environment and the trivial identity assignment, by repeatedly performing the following steps, until none applies.

(1) Remove redundant statements $\bullet \le \bullet$, $\bullet \le \times$, $\times \le \times$, $a \le \times$, $\bullet \le a$.

(2) Eliminate indirect assignments $a \le \bullet$, $\times \le a$ by substituting $\bullet$ (respectively $\times$) for a and extending the attribute assignment with $a := \bullet$ or $a := \times$ respectively.

Say the result is a reduced environment Γ^- and an attribute assignment $\diamond^-(\Gamma)$. Now Γ is consistent if and only if $(\times \le \bullet) \notin \Gamma^-$. The above procedure always terminates, so the consistency property is decidable.

All this can be used to construct an attribution (if it exists) of a given conventional typing $\mathcal{T}$ for g. First construct a general attributed version $\mathcal{U}_0$ of $\mathcal{T}$ by attributing each subtype with a fresh attribute variable. Then construct $\Gamma_0 = \Gamma^{\mathcal{E}}(g,\mathcal{U}_0)$. If Γ_0 is inconsistent then $\mathcal{T}$ is not attributable. If, on the other hand, Γ_0 is consistent then $\langle \mathcal{U}_0, \Gamma_0\rangle$ is the *principal attributed version* of $\mathcal{T}$ (i.e., the least restrictive one: all other attributed versions are obtained by instantiating attribute variables).

Simplified attributions

The above reduction method for attribute environments can be used to give a simplified typing, i.e. a typing with a minimal number of attribute variables and coercion statements.

Given Γ, first determine Γ^- and $\diamond^-(\Gamma)$. Now we simplify Γ^- and extend $\diamond^-(\Gamma)$ by repeated application of the following step.

(3) Remove cycles $a \le a_1, a_1 \le a_2, \ldots, a_{n-1} \le a_n$, substitute a for $a_1, \ldots, a_n$ in the remaining statements, and add $a_1 := a, \ldots, a_n := a$ to the attribute assignment.

The resulting environment is denoted by $\min(\Gamma)$, and the final attribute assignment by $\diamond(\Gamma)$.

If $\langle \mathcal{U}, \Gamma\rangle$ is typing for g, then $\langle \mathcal{U}^{\diamond(\Gamma)}, \min(\Gamma)^{\diamond(\Gamma)}\rangle$ is its simplified version. The examples in Section 6 are given in this simplified form.

Combined type inference

Type inference in *Clean* can now be described as follows. Given a graph, first determine its principal conventional type $\mathcal{T}_0$. Then apply the principal attribution algorithm to determine (if possible) a principal attribution for $\mathcal{T}_0$. Finally use the simplification procedure to obtain a readable result.

6. Examples

In this section we will give some examples illustrating the use of uniqueness typing in practice. All programs are written in *Clean* which has a Haskell-like syntax. For obvious reasons it was impossible to adopt the notation for uniqueness types introduced in the present paper. In *Clean* one can assign a uniqueness attribute, say `a`, to a (sub)type by putting `a:` in front of it. The unique attribute $\bullet$ is denoted as `*`. By convention, the attribute variables are chosen from `a`,`b`,`a1`,....

Moreover, there is no denotation for $\times$; this attribute is chosen by default. Lastly, coercion statements expressing internal consistency of types are left implicit. E.g. $\mathsf{List}^b(\alpha^a)$ $[b \leq a]$ is simply denoted as b:[a:x] (the square brackets in the latter type denote the list constructor; x is a conventional type variable).

The first example shows how a function can be defined that copies the contents of a file to another file using the following I/O primitives.

```
FWriteC :: Char *File → *File
FReadC :: File → (Bool, Char, File)
```

FWriteC requires a unique input file which is updated destructively. The output is also unique, to prepare for further manipulations. FReadC is a non-destructive function. Its output consists of three parts: a boolean indicating success or failure of the read operation, the character that has been read from the input file, and the file that results from the input file after deleting the first character. The function FCopy is defined straightforwardly.

```
FCopy src dst | ok  = FCopy nsrc (FWriteC char dst)
                    = dst
                    where
                    (ok, char, nsrc) = FReadC src
```

The compiler infers FCopy :: File *File → *File.

In the second example, a function is given to append one list to another. To examine the effect on its uniqueness typing, we specify two distinct versions of append.

```
append [] li         = li
append (hd : tl) li = hd : append tl li
```

The type derived by the type system is

```
append :: b:[a:x] c:[a:x] → d:[a:x], c ≤ d
```

The coercion statement specifies that spine uniqueness of the result list (d = *) depends only on the spine attribute c of the second argument.

As mentioned in Wadler (1990), storage use can be optimized whenever the first list argument is spine unique. In contrast to Wadler (1990), the recursive call of append cannot be evaluated immediately, since (a part of) the result of this function is likely to be discarded by a context in which append is applied. However, if it is guaranteed that the spine of the list is always evaluated in any context, a smart compiler might produce an efficient loop version of append in which no auxiliary storage at all is needed.

An alternative definition of append can be obtained by using the well-known function foldr:

```
foldr op zero []         = zero
foldr op zero (hd : tl)  = op hd (foldr op zero tl)
```

The uniqueness type of foldr (as derived by the type system) is

```
foldr :: (a:x → d:(b:y → b:y)) b:y c:[a:x] → b:y
```

(Observe that the attribute of the second → is d.) The only essential restriction on applications of foldr w.r.t. uniqueness of arguments comes from the fact that the first argument op is used twice in the right-hand size of foldr. This results in the (default) non-unique attribute at the first → constructor indicating that foldr may not be applied to functions to which only one reference is allowed (see Barendsen and Smetsers (1993a) for further explanation).

Now one can define append as follows.

```
append l1 l2 = foldr (:) l2 l1
```

The derived uniqueness type of append is the same as the type derived for the previous version. The equality of both types (in particular with respect to uniqueness information) suggests an equivalent operational behaviour: the results are constructed in essentially the same way, namely by copying the first list and redirecting the final tail pointer to the second list (which is the most obvious way to append two lists). This shows that the derived uniqueness properties give some additional information on append. E.g., if one accidently exchanges both list arguments in the application of foldr, i.e.

```
append l1 l2 = foldr (:) l1 l2
```

the conventional type remains the same; the uniqueness type, however, becomes different. In general, one can use uniqueness typing not only to produce more efficient code but also for a partial correctness check of programs.

The final example has been taken from Guzmán and Hudak (1990). It shows a function mark for marking a graph, e.g. to avoid that nodes of a graph are visited more than once during its traversal. The definition of mark as it can be found in Guzmán and Hudak (1990) is as follows.

```
mark g loc =
  if g == empty || g [loc,0] = True
  then g
  else let* left = g [loc,1]
            right = g [loc, 2]
            g' = update! g [loc,0] True
       in mark (mark g' left) right
```

Guzmán and Hudak (1990) infer the following type for mark.

$$\mathsf{mark} : \mathsf{Array}(\iota, \mathsf{Int}) \xrightarrow{ws} \iota \xrightarrow{r} \mathsf{Array}(\iota, \mathsf{Int}),$$

implying that all updates can be performed destructively.

In the remainder of this section, we will derive a version of mark in *Clean* with similar uniqueness properties. This will be done in three steps. The first version we consider is obtained by a more or less direct translation of the original mark function. To represent graphs, we introduce the following auxiliary synonym types.

```
:: Graph     :== Array Node
:: Node      :== (Marking, Arity, Arguments)
:: Marking   :== Bool
:: Arity     :== Int
:: Arguments :== Array Int
```

The definition of mark in *Clean* is as follows.

```
mark g loc | marking
  = g
  = m_args new_g arity args
  where
  (marking,arity,args) = lookup g loc
  new_g = update g loc (True,arity,args)

  m_args g 0 args    = g
  m_args g n args    = m_args g (lookup args n) (n - 1) args
```

For the sake of clarity, we prefer to denote array selection as an application of a function called lookup. Now suppose lookup and update are typed as follows.

```
lookup :: b:(Array a:x) Int → a:x
update :: *(Array a:x) Int a:x → * Array a:x
```

Then the function mark will be rejected by the type system. This is due to the fact that g is used twice in the right-hand side of mark. Consequently, both references to g are $\otimes$-marked, thus preventing unique usage of g.

Note that in the original version the lookup and update operations are evaluated in a certain order (depending on the special semantics of the let * construct). One can also enforce sequentialisation by defining a slightly modified version of lookup

```
lookup :: b:(Array x) Int → (x, b:(Array x))
```

Besides the selected element, lookup yields the array from which selection took place as a result. This array can be used for further manipulations (similar to the files in the results of the file operations mentioned in the FCopy example). Obviously, the elements of the array are no longer unique.

The second definition of mark, including its uniqueness type, is

```
mark :: b:(Array x) Int → b:(Array x)
mark g loc | marking
  = g
  = m_args upd_g arity args
  where
  (marking,arity,args) = node
  (node,look_g)    = lookup g loc
  upd_g            = update look_g loc (True,arity,args)
```

The final version of mark that even leaves the uniqueness of nodes intact can be obtained by using a new array operation, called exchange.

```
exchange :: b:(Array a:x) Int a:x → (a:x, b:(Array a:x))
```

This operation is based on the observation that one can safely select an element from an array without destroying uniqueness properties of this array if a new (unique) element is inserted at the place of the selected one. Note that this is essentially what the uniqueness type of exchange indicates.

This leads to the following definition for mark

```
mark :: b:(Array a:x) Int → b:(Array a:x)
mark g loc | marking
  = g
  = m_args new_g arity args
  where
  (marking,arity,args) = node
  (node,exc_g)     = exchange g loc empty_node
  (empty, new_g    = exchange look_g loc (True,arity,args)
```

First note that the resulting graph is fully unique whenever the input graph is also fully unique. Moreover, the storage cell containing the constructor of the node being marked becomes garbage. Obviously one can recycle this cell immediately for building the new marked node instead of returning it to free storage. In fact, one can even reuse most of its old contents; the only part that has to be updated is the mark field. Lastly, it requires only a simple analysis to derive that the node exchanged with the selected node is returned by the second call of exchange. Hence, it suffices to allocate only one such node during program execution.

7. Conclusions and Related Work

We have developed a very powerful type system to characterize reference structures in graphs, in order to express uniqueness constraints of function arguments. In the present paper, polymorphism has been extended to uniqueness attributes. The resulting system turns out to be decidable.

The system has been implemented as part of the *Clean* compiler. The applicability is not restricted to languages whose *syntax* is based on graph rewriting. In fact, the type system is also suited for other functional languages, since most of them use graph-like structures in their *implementations*.

The weighted reference count analysis has been inspired by Guzmán and Hudak (1990). This paper addresses the mutability 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.

In Wadler (1990), a type system including linear types is developed. The paper also uses Wadsworth's lambda reduction. The relation between linear and uniqueness typing can be summarized as follows.

uniqueness typing	=	pure linear typing
	+	conventional typing
	+	subtyping
	+	strategy-aware reference analysis.

Based on the idea of uniqueness typing, Jacobs (1993) developed a *logical* system explicitly mixing conventional and linear constructive logic.

In Sastry et al. (1993), the update problem is addressed by determining a safe evaluation order via abstract interpretation. It would be interesting to investigate how our stragegy-dependent reference analysis can be refined using their techniques. This would lead to a more liberal notion of typing.

References

Achten, P.M. and M.J. Plasmeijer (1995). The ins and outs of Clean I/O, *Journal of Functional Programming*. To appear.

Achten, P.M., J.H.G. van Groningen and M.J. Plasmeijer (1993). High level specification of I/O in functional languages, *Proceedings of the International Workshop on Functional Languages*, Lecture Notes in Computer Science, Springer-Verlag, Berlin.

van Bakel, S.J., J.E.W. Smetsers and S. Brock (1992). Partial type assignment in left-linear term rewriting systems, *in:* J.C. Raoult (ed.), *Proceedings of the 17th Colloqium on Trees and Algebra in Programming (CAAP'92)*, Rennes, France, Lecture Notes in Computer Science 581, Springer-Verlag, Berlin, pp. 300–322.

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

Barendsen, E. and J.E.W. Smetsers (1992). Graph rewriting and copying, *Technical Report 92-20*, Computing Science Institute, University of Nijmegen.

Barendsen, E. and J.E.W. Smetsers (1993a). Conventional and uniqueness typing in graph rewrite systems, *Technical Report CSI-R9328*, Computing Science Institute, University of Nijmegen.

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

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

Girard, J.-Y., Y. Lafont and P. Taylor (1989). *Proofs and Types*, Cambridge Tracts in Theoretical Computer Science 7, Cambridge University Press.

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

Jacobs, B.P.F. (1993). Personal communication.

Plasmeijer, M.J. and M.C.J.D. van Eekelen (1995). Concurrent Clean 1.0 language report, Computing Science Institute, University of Nijmegen, To appear.

Sastry, W., A.V.S. Clinger and Z.M. Ariola (1993). Order-of-evaluation analysis for destructive updates in strict functional languages with flat aggregates, *Proceedings of the Conference on Functional Languages and Computer Architectures (FPCA)*, Copenhagen, Denmark, ACM Press, pp. 266–276.

Troelstra, A.S. (1992). *Lectures on Linear Logic*, CSLI Lecture Notes 29, CSLI, Stanford.

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

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