Math. Struct. in Comp. Science (1996), *vol.* 00, *pp.* 1–36

Uniqueness Typing for Functional Languages with Graph Rewriting Semantics

ERIK BARENDSEN and SJAAK SMETSERS

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

Received November 1995, revised May 1996

We present two type systems for term graph rewriting: conventional typing and (polymorphic) uniqueness typing. The latter is introduced as a natural extension of simple algebraic and higher-order uniqueness typing. The systems are given in natural deduction style using an inductive syntax of graph denotations with familiar constructs such as let and case.

The conventional system resembles traditional Curry-style typing systems in functional programming languages. Uniqueness typing extends this with reference count information. In both type systems, typing is preserved during evaluation, and types can be determined effectively. Moreover, with respect to a graph rewriting semantics, both type systems turn out to be sound.

Contents

1. Introduction

In recent years, various proposals have been brought up to capture the notion of assignment in a functional context. This desire is paradoxical, because the absence of side effects is one of the main reasons why functional programming languages are often praised. As a consequence of this absence, functional languages have the fundamental property of

referential transparency: each (sub)expression denotes a fixed value, independently of the way this value is computed.

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

There is a solution for this problem which can be achieved entirely within the functional framework: by delivering a sequence of instructions for the operating system as a result of a functional expression. One could call this a *delegating* approach, since the computation only *prepares* for the external execution of, for example, I/O tasks. In the literature, this method is known as *stream based* I/O. An application can be found in the language *Haskell*.

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

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

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

Uniqueness typing can be regarded as a combination of linear typing (dealing with unique objects) and traditional typing (for non-unique objects), connected by a subtyping mechanism. In fact, the part handling uniqueness allows discarding of objects, so it corresponds more closely to *affine logic*, see Blass (1992). A logical/categorical proposal for a related combination appears in Benton (1994).

The present paper describes the uniqueness type system in natural deduction style, using an inductive syntax for graph expressions. The emphasis on graph *denotations* contrasts the original presentation, which referred directly to the node/reference structure of (non-inductive) graph *objects*. The graph syntax is similar to the object language in the equational approach towards Term Graph Rewriting of Ariola and Klop (1995). The operational semantics of the object language is given by the concept of Term Graph Rewriting, as introduced by Barendregt et al. (1987). Each expression is translated into a term graph. In contrast with Ariola and Klop (1995), we refrain from defining a reduction relation on the expressions directly.

The paper is organized as follows. After a very short and informal introduction to Term Graph Rewriting (Section 2), we introduce a formal language for denoting graph expressions and function definitions (Section 3). Section 4 describes the incorporation of higher-order functions in our system. In Section 5 a Curry style (conventional) type system is introduced. The system is not new, but the terminology and techniques in this section prepare for the development of uniqueness typing, which will proceed along the same lines. We prove preservation of typing during reduction and the existence of principal types. After an informal introduction to uniqueness typing in Section 6, Section 7 describes a simple (i.e., non-polymorphic) algebraic uniqueness type assignment system. This system is extended via higher-order typing to the complete polymorphic uniqueness type system (Section 8). The results for the conventional system are extended to uniqueness typing. At the end of Section 8 we describe how uniqueness type inference proceeds in the functional programming language *Clean* (see also Plasmeijer and van Eekelen (1995). We conclude with a discussion of related work and future research (Section 9).

The original uniqueness type system is rather complex, particularly due to the refined reference analysis. To avoid that the reader gets entangled in technical details, this analysis is kept here as simple as possible: it does not take the evaluation order into account. Due to the present formalization, the system can easily be compared with other proposals based on linear and affine logic.

This paper is an elaborated version of the work presented in Barendsen and Smetsers (1995a) and Barendsen and Smetsers (1995c).

2. Term graph rewriting

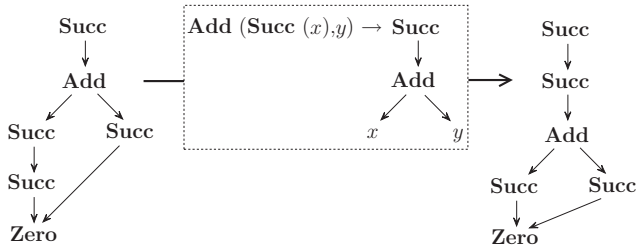
Term graph rewrite systems (TGRS's) have been introduced in Barendregt et al. (1987), see also Barendsen and Smetsers (1994). This section summarizes some basic concepts.

Term graph rewriting can be seen as an extension of term rewriting with sharing. In term rewrite rules, multiple occurrences of variables lead to duplication of actual instances. In TGRS's, such duplications are avoided by copying references to the objects instead of copying the objects themselves.

The objects in TGRS's are directed graphs in which each node is labelled with a *symbol*. Each symbol $\mathbf{S}$, say, has a fixed arity determining the number of outgoing edges (references to the arguments) of any node labelled with $\mathbf{S}$.

We distinguish two kinds of symbols: function symbols and algebraic constructor symbols. Function symbols are introduced by rewrite rules which specify transformations of graphs. Each rewrite rule $\mathbf{F}\vec{p} \to g$ consists of left-hand side $\mathbf{F}\vec{p}$ (the *pattern*) and a right-hand side g (the *result*). Both $\mathbf{F}\vec{p}$ and g are graphs. We say that a rule is *left-linear* if the pattern $F\vec{p}$ is a tree (hence $\vec{p}$ does not contain multiple occurrences of the same variable). This enables us to denote left-hand sides of rewrite rules as terms. We consider *function/constructor rules*: $\vec{p}$ contains only variables and algebraic constructors.

The following picture shows a graph rewrite step according to the displayed rule.



Let $\mathcal{R}$ be a set of rewrite rules. The (multistep) rewrite relation induced by $\mathcal{R}$ is denoted by $\underset{\mathcal{R}}{\twoheadrightarrow}$.

Algebraic constructors are assumed to introduced by a so-called *algebraic type system* $\mathcal{A}$ containing specifications like

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

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

3. Graph denotations

Syntax

In order to present our type systems in natural deduction style, we introduce *graph denotations* generated by an inductive syntax. The language constructs reflect the essential aspects of graph rewriting: application, sharing (implicitly, by multiple occurrences of the same variable, and explicitly, using let), cycles (sharing using letrec), pattern matching (case), and function definitions.

Instead of admitting several rules (one for each alternative pattern) for functions, we collect all alternatives in a case construct.

Definition 3.1 (i) The objects are expressions generated by the following abstract syntax.

$$\begin{array}{lcl} E & ::= & x \mid \mathbf{S}(E_1, \ldots, E_k) \mid \mathsf{let}\ x = E\ \mathsf{in}\ E' \mid \mathsf{letrec}\ \vec{x} = \vec{E}\ \mathsf{in}\ E' \mid \mathsf{case}\ E\ \mathsf{of}\ \vec{P}|\vec{E}, \\ P & ::= & \mathbf{C}(x_1, \ldots, x_k). \end{array}$$

Here $x, \vec{x}$ range over (sequences of) term variables, and **S** over some set of *symbols* of fixed arity (we will suggestively use **F** for functions and **C** for data constructors). The set of free variables of E (notation $\mathrm{FV}(E)$) is defined as usual.

(ii) Function definitions are expressions of the form

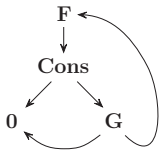
$$\mathbf{F}(x_1, \ldots, x_k) = E.$$

Some hygiene with respect to variables is necessary: in $\mathsf{let}\ x = E\ \mathsf{in}\ E'$, the variable x should not occur free in E. We only consider *left-linear* functions: no variable occurs more than once in the same pattern. Moreover, these pattern variables are 'local': they occur only in $\vec{P}|\vec{E}$.

Example 3.2 (i) The expression

$$\mathsf{let}\ x = \mathbf{0}\ \mathsf{in}\ \mathsf{letrec}\ z = \mathbf{F}(\mathbf{Cons}(x, \mathbf{G}(x, z)))\ \mathsf{in}\ z$$

denotes the following graph.



(ii) Addition on the algebraic type of natural numbers

$$\mathsf{Nat} = \mathbf{0} \mid \mathbf{S}(\mathsf{Nat})$$

can be expressed by

$$\begin{array}{lllll} \mathbf{Add}(x,y) = \mathsf{case}\ x\ \mathsf{of} & \mathbf{0} & \mid & y \\ & \mathbf{S}(x') & \mid & \mathbf{S}(\mathbf{Add}(x',y)). \end{array}$$

Semantics

We now give a formal account of the (intuitively clear) interpretation of expressions and function definitions as graphs and rewrite rules respectively. Instead of using drawings, like above, or 4-tuples (see Barendregt et al. (1987) and Barendsen and Smetsers (1994)), we specify graphs in an equational style (cf. Barendregt et al. (1987), Ariola and Klop (1995)). Each equation is a *node specification* of the form

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

where $n, n_1, \ldots, n_k$ are variables. Moreover, the topmost node (root) is indicated explicitly. E.g., the graph in Example 3.2 (i) can be denoted by

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

Definition 3.3 A *graph* is a tuple $g = \langle r \mid G \rangle$. The *variable set* of g (notation $\mathrm{V}(g)$) is the collection of variables appearing in r, G. The set of *free variables* of g (notation $\mathrm{FV}(g)$) consists of those in $\mathrm{V}(g)$ that do not appear as the left-hand side of an equation in G; the other (*bound*) variables are indicated by $\mathrm{BV}(g)$. We will identify graphs that only differ in the names of bound variables.

Using this formalism, it is easy to define the interpretation of expressions as graphs.

Our type systems for expressions deal with the full syntax of Definition 3.1. However, to provide a simple and direct translation to graph rewrite systems we only consider case expressions in function definitions at the topmost level, distinguishing cases as to one of the input variables. This is no serious restriction: it is always possible to introduce auxiliary functions to express nested pattern matching.

Definition 3.4 For each expression E, the *graph interpretation* of E (notation $[\![E]\!]$) is defined inductively as follows. The denotation $[\vec{x} := \vec{r}]$ stands for simultaneous substitution of $\vec{r}$ for (the free occurrences of) $\vec{x}$.

$$\begin{aligned}
[\![x]\!] &= \langle x \mid \emptyset \rangle, \\
[\![\mathbf{S}\vec{E}]\!] &= \langle r \mid \{r = \mathbf{S}(\vec{r})\} \cup \vec{G} \rangle \quad \text{where } \langle r_i \mid G_i \rangle = [\![E_i]\!] \text{ and } r \text{ fresh}, \\
[\![\mathsf{let}\ x = E\ \mathsf{in}\ E']\!] &= \langle r' \mid G \cup G' \rangle [x := r], \\
[\![\mathsf{letrec}\ \vec{x} = \vec{E}\ \mathsf{in}\ E']\!] &= \langle r' \mid \vec{G} \cup G' \rangle [\vec{x} := \vec{r}].
\end{aligned}$$

We have omitted the case construct: it will be treated in the translation of function definitions. Note that our interpretation is such that $\mathrm{FV}(E) = \mathrm{FV}([\![E]\!])$.

The interpretation of expressions naturally gives rise to an equivalence relation on these expressions.

Definition 3.5 The expressions E and E' are *graph equivalent* (notation $E \sim E'$) if $[\![E]\!] = [\![E']\!]$.

Once we have interpreted expressions we can interpret function definitions.

Definition 3.6 (i) The *graph interpretation* of a function definition $\mathbf{F}\vec{x} = E$ is specified as follows. We distinguish two cases:

Case 1. No pattern matching: E is case free. Then the interpretation consists of a single rewrite rule:

$$[\![\mathbf{F}\vec{x} = E]\!] = \mathbf{F}\vec{x} \to [\![E]\!].$$

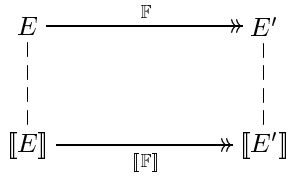
Case 2. Pattern matching. Then the interpretation is a collection of rules:

$$[\![\mathbf{F}(x, \vec{y}) = \mathsf{case}\ x\ \mathsf{of}\ \vec{P} \mid \vec{E}]\!] = \begin{cases} \mathbf{F}(P_1, \vec{y}) \to [\![E_1]\!], \\ \qquad\qquad \vdots \\ \mathbf{F}(P_n, \vec{y}) \to [\![E_n]\!]. \end{cases}$$

(ii) Let $\mathbb{F}$ be a set of function definitions. Then $[\![\mathbb{F}]\!]$ is the collection of graph rewrite rules obtained from $\mathbb{F}$-elements.

The function definitions induce a computation relation on expressions.

Definition 3.7 Let $\mathbb{F}$ be a collection of function definitions. Then we write $E \twoheadrightarrow_{\mathbb{F}} E'$ if $[\![E]\!] \underset{[\![\mathbb{F}]\!]}{\twoheadrightarrow} [\![E']\!]$.



For a formal definition of graph rewriting the reader is referred to Barendregt et al. (1987) (see also Barendsen and Smetsers (1994)).

4. Higher-order functions

In term graph rewriting all symbols have a fixed arity. Moreover, there is no abstraction operator (like λ) to construct anonymous functions. This prevents the use of *functions* as arguments or as results.

The usual way (in functional programming languages) to incorporate higher-order functions is to admit partial (often called *Curried*) applications. These are written as

$$\mathbf{F}E_1 \cdots E_k$$

(with $k < \text{arity}(\mathbf{F})$), denoting the function $\lambda\vec{x}.\mathbf{F}(E_1, \ldots, E_k, x_{k+1}, \ldots, x_{\text{arity}(\mathbf{F})})$.

Partial applications can be simulated in graph rewriting by introducing auxiliary symbols $\mathbf{F}_k$ and writing $\mathbf{F}E_1 \cdots E_k$ as

$$\mathbf{F}_k(E_1, \ldots, E_k)$$

(so $\mathbf{F}_{\text{arity}(\mathbf{F})} = \mathbf{F}$). Moreover, we add an application operator Ap to our syntax which collects arguments, i.e. (roughly),

$$\mathsf{Ap}(\mathbf{F}_k(\vec{E}), E') = \mathbf{F}_{k+1}(\vec{E}, E').$$

See Barendsen and Smetsers (1993) for a detailed description of the graph rewriting semantics.

We will present our type systems first for the traditional, purely algebraic, syntax and then describe the type-theoretic extension dealing with partial applications. For conventional typing this extension is straightforward. In the case of uniqueness typing, however, this turns out to be a subtle matter.

5. Conventional Typing

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

Syntax

Definition 5.1 *Types* are built up from type variables and type constructors.

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

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

The function space type constructor $\to$ is used when dealing with higher-order functions. We will treat these at the end of this section.

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

Definition 5.2 A *symbol type* of arity k is a $k+1$ tuple $(\sigma_1, \ldots, \sigma_k, \tau)$. We will sugestively denote this as

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

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

Definition 5.3 (i) Let $\mathcal{A}$ be an algebraic type system. The specifications in $\mathcal{A}$ give the types of the algebraic constructors. Let

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

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

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

For example, for lists one has

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

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

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

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

$$\mathcal{F} \vdash \mathbf{F} : \vec{\sigma} \rhd \tau.$$

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

$$\frac{\mathcal{F}, \mathcal{A} \vdash \mathbf{S} : \vec{\sigma} \rhd \tau}{\mathcal{F}, \mathcal{A} \vdash \mathbf{S} : \vec{\sigma}[\alpha := \rho] \rhd \tau[\alpha := \rho]}$$

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

Definition 5.4 (i) A *basis* is a finite set of declarations of the form $x{:}\tau$ concerning distict variables.

(ii) The type system deals with *typing statements* of the form

$$B \vdash E : \sigma,$$

where B is a basis. Such a statement is valid if it can be produced using the following derivation rules. Below we abbreviate $B \cup \{x{:}\sigma\}$ by $B, x{:}\sigma$.

$$B, x{:}\sigma \vdash x : \sigma \quad \text{(variable)}$$

$$\frac{\mathcal{F}, \mathcal{A} \vdash \mathbf{S} : \vec{\sigma} \rhd \tau \qquad B \vdash \vec{E} : \vec{\sigma}}{B \vdash \mathbf{S}\vec{E} : \tau} \text{(application)}$$

$$\frac{B \vdash E : \sigma \qquad B, x{:}\sigma \vdash E' : \sigma'}{B \vdash \mathsf{let}\ x = E\ \mathsf{in}\ E' : \sigma'} \text{(sharing)}$$

$$\frac{B, \vec{x}{:}\vec{\sigma} \vdash E_i : \sigma_i \qquad B, \vec{x}{:}\vec{\sigma} \vdash E' : \tau}{B \vdash \mathsf{letrec}\ \vec{x} = \vec{E}\ \mathsf{in}\ E' : \tau} \text{(cycle)}$$

$$\frac{B \vdash E : \tau \qquad \mathcal{A} \vdash \mathbf{C}_i : \vec{\sigma_i} \rhd \tau \qquad B, \vec{x_i}{:}\vec{\sigma_i} \vdash E_i : \tau'}{B \vdash \mathsf{case}\ E\ \mathsf{of}\ \vec{P}|\vec{E} : \tau' \quad (\text{if } P_i = \mathbf{C}_i\vec{x_i})} \text{(pattern matching)}$$

(iii) A function definition $\mathbf{F}\vec{x} = E$ is called *type correct* if

$$\vec{x} : \vec{\sigma} \vdash E : \tau$$

where $\vec{\sigma} \rhd \tau$ is the standard type of $\mathbf{F}$. A collection of function definitions is type correct if its members are type correct in the above sense.

Semantics

We will show that the type system is sound and complete with respect to graph rewriting. We first recapitulate the notion of typing for graphs and rewrite rules from Barendsen and Smetsers (1993). This notion is a graph-theoretic variant of the typing concept presented in van Bakel et al. (1992) for Term Rewrite Systems.

Definition 5.5 (TGRS Graph typing) Let $g = \langle r \mid G \rangle$ be a graph. A *typing* for g is a function $\mathcal{T}$ assigning a type to each element of $\mathrm{V}(g)$ such that for any equation $x = \mathbf{S}(\vec{y})$ in G

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

That is, $\mathcal{T}$ is a type assignment to nodes that satisfies 'local correctness'. We say that $\mathcal{T}$ *types g with σ* (notation $\mathcal{T}(g) = \sigma$) if furthermore $\mathcal{T}(r) = \sigma$. Moreover, g *is typable with σ* (notation $g : \sigma$) if there exists $\mathcal{T}$ with $\mathcal{T}(g) = \sigma$.

Now we can give a semantic notion of expression typing.

Definition 5.6 Let E be an expression.

(i) We say that $\mathcal{T}$ *types E with σ* (notation $\mathcal{T} \models E : \sigma$) if $\mathcal{T}([\![E]\!]) = \sigma$.

(ii) This notation is extended to sequences (in particular, bases): $\mathcal{T} \models B$ iff $\mathcal{T} \models x : \sigma$ for each $(x : \sigma) \in B$.

(iii) By $B \models E : \sigma$ we denote that B is extendible to a typing of $[\![E]\!]$, i.e., for some $\mathcal{T}$ one has

$$\mathcal{T} \models B \text{ and } \mathcal{T} \models E : \sigma.$$

Proposition 5.7 (Soundness of expression typing) *For any* B, E, σ

$$B \vdash E : \sigma \;\Rightarrow\; B \models E : \sigma.$$

Proof. By induction on the derivation of $B \vdash E : \sigma$. □

Proposition 5.8 (Completeness of expression typing) *For any* B, E, σ

$$B \models E : \sigma \;\Rightarrow\; B \vdash E : \sigma.$$

Proof. By induction on the structure of E. □

Corollary 5.9 $B \vdash E : \sigma,\ E \sim E' \;\Rightarrow\; B \vdash E' : \sigma$

Typing of graph rewrite rules can be formulated as follows.

Definition 5.10 (TGRS Rule typing) (i) A type assignment $\mathcal{T}$ to variables can be extended to algebraic *patterns* in a straightforward way:

$$\mathcal{F}, \mathcal{A} \vdash \mathbf{C} : \vec{\rho} \rhd \sigma,\ \mathcal{T}(\vec{p}) = \vec{\rho} \;\Rightarrow\; \mathcal{T}(\mathbf{C}\vec{p}) = \sigma.$$

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

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

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

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

Proposition 5.11 (Soundness of function typing)

$$\mathbb{F} \textit{ is type correct } \Rightarrow\; [\![\mathbb{F}]\!] \textit{ is type correct.}$$

Proof. Straightforward. □

Subject Reduction

The following is proved in Barendsen and Smetsers (1993).

Theorem 5.12 (TGRS Reduction typing) *Suppose* $\mathcal{R}$ *is type correct. Then*

$$\left.\begin{array}{r} g : \sigma \\ g \xrightarrow[\mathcal{R}]{} g' \end{array}\right\} \quad \Rightarrow \quad g' : \sigma.$$

Proof. [Sketch] If a rewrite rule (say with typing $\mathcal{T}$, cf. Definition 5.10) applies to a certain typed graph g, then there is a substitution $*$ such that $\mathcal{T}^*$ (on the pattern of the rule) corresponds to the typing of the matching part of g. Now $\mathcal{T}^*$ (on the result of the rule) can be used to type the contractum. □

By our soundness and completeness results, the so-called Subject Reduction Property is an easy corollary of the previous theorem.

Subject Reduction Theorem 5.13 *Suppose* $\mathbb{F}$ *is type correct. Then*

$$\left.\begin{array}{l} B \vdash E : \sigma \\ E \twoheadrightarrow_{\mathbb{F}} E' \end{array}\right\} \quad \Rightarrow \quad B \vdash E' : \sigma.$$

Type Inference

In this section we will show that type assignment has the *principal type property*: if an expression E is typable then there exists a most general typing for E. The presentation below has been inspired by Barendregt (1992). The idea of strictly splitting type reconstruction into generation of equations and unification is due to Wand (1987). The present algorithm is an inductive variant of the procedure on graphs described in Barendsen and Smetsers (1993).

Definition 5.14 (i) A *type equation* is an expression of the form $\sigma \simeq \tau$.

(ii) Let $\mathcal{E} = \{\sigma_1 \simeq \tau_1, \ldots, \sigma_n \simeq \tau_n\}$ be a finite set of type equations. A *solution* for $\mathcal{E}$ is a substitution $*$ such that

$$\sigma_1^* = \tau_1^*, \ldots, \sigma_n^* = \tau_n^*.$$

In that case one writes $* \models \mathcal{E}$. The notion of *most general solution* for $\mathcal{E}$ is defined as usual.

Unification Theorem 5.15 *There exists a recursive function* Unify *having as input finite sets of equations between types such that*

$$\begin{array}{rcl} \mathcal{E} \textit{ has a solution} & \Rightarrow & \mathrm{Unify}(\mathcal{E}) \textit{ is the most general solution for } \mathcal{E}; \\ \mathcal{E} \textit{ has no solution} & \Rightarrow & \mathrm{Unify}(\mathcal{E}) = \mathsf{fail}. \end{array}$$

Proof. See Robinson (1965). □

The next step is to associate with each expression E a set of type equations in such a way that typability of E can be formulated in terms solvability of those equations.

Definition 5.16 Let $*, *'$ be substitutions.

(i) Let σ be a type. Then $*, *'$ are *equivalent* with respect to σ (notation $* \sim_\sigma *'$) if $*(\alpha) = *'(\alpha)$ for all α occurring in σ.

(ii) This notion is extended sequences of bases and/or types: we write $* \sim_{B,\sigma} *'$ if $* \sim_\tau *'$ for all types τ appearing in B, σ.

Definition 5.17 Let E be an expression, B a basis, and σ a type. A set of type equations $\mathcal{E}$ is *exact* for B, E, σ if for each substitution $*$ one has

(1) $* \models \mathcal{E} \quad \Rightarrow \quad B^* \vdash E : \sigma^*$

(2) $B^* \vdash E : \sigma^* \quad \Rightarrow \quad *' \models \mathcal{E}$ for some $*'$ with $*' \sim_{B,\sigma} *$.

Proposition 5.18 *Let E be an expression, B be a basis, and $\sigma \in \mathbb{T}$. Then there exists a finite set of equations $\mathcal{E} = \mathcal{E}(B, E, \sigma)$, computable from B, E, σ, such that $\mathcal{E}$ is exact for B, E, σ.*

Proof. Define $\mathcal{E}(B, E, \sigma)$ by induction on E. To avoid notational confusion we write $B \cup x{:}\sigma$ instead of $B, x{:}\sigma$.

$$
\begin{array}{rcl}
\mathcal{E}(B, x, \sigma) & = & \{B(x) \simeq \sigma\} \quad \text{(regard } B \text{ as a partial function)}, \\
\mathcal{E}(B, \mathbf{S}\vec{E}, \sigma) & = & \bigcup_i \mathcal{E}(B, E_i, \tau_i) \cup \{\rho \simeq \sigma\} \\
 & & \text{where } \vec{\tau} \rhd \rho \text{ is the standard } \mathcal{F}, \mathcal{A}\text{-type of } \mathbf{S}, \\
\mathcal{E}(B, \mathsf{let}\ x = E\ \mathsf{in}\ E', \sigma) & = & \mathcal{E}(B, E, \alpha) \cup \mathcal{E}(B \cup x{:}\alpha, E', \sigma) \\
 & & \alpha \text{ fresh}, \\
\mathcal{E}(B, \mathsf{letrec}\ \vec{x} = \vec{E}\ \mathsf{in}\ E', \sigma) & = & \mathcal{E}(B \cup \vec{x}{:}\vec{\alpha}, E', \sigma) \cup \bigcup_i \mathcal{E}(B \cup \vec{x}{:}\vec{\alpha}, E_i, \alpha_i) \\
 & & \vec{\alpha} \text{ fresh}, \\
\mathcal{E}(B, \mathsf{case}\ E\ \mathsf{of}\ \vec{P}|\vec{E}, \sigma) & = & \mathcal{E}(B, E, \alpha) \cup \bigcup_i (\mathcal{E}(B \cup \vec{x_i}{:}\vec{\sigma_i}, E_i, \sigma) \cup \{\tau_i \simeq \alpha\}) \\
 & & \text{if } P_i = \mathbf{C}_i\vec{x_i} \text{ and the standard type of } \mathbf{C}_i \text{ is } \vec{\sigma_i} \rhd \tau_i, \\
 & & \alpha \text{ fresh}.
\end{array}
$$

One shows (1) by induction on E and (2) using induction on the derivation of $B^* \vdash E : \sigma^*$. □

Definition 5.19 Let E be an expression. The pair (B, σ) is a *principal typing* for E if

(1) $B \vdash E : \sigma$,

(2) $B' \vdash E : \sigma' \quad \Rightarrow \quad B' \supseteq B^*,\ \sigma' = \sigma^*$ for some substitution $*$.

Principal Typing Theorem 5.20 *There exists a recursive function pt such that*

$$
\begin{array}{rcl}
E \text{ is typable} & \Rightarrow & pt(E) \text{ is a principal typing for } E; \\
E \text{ is not typable} & \Rightarrow & pt(E) = \mathsf{fail}.
\end{array}
$$

Proof. Say $\mathrm{FV}(E) = \{x_1, \ldots, x_k\}$. Set $B_0 = \{x_1{:}\alpha_1, \ldots, x_k{:}\alpha_k\}$, and $\sigma_0 = \alpha$ (all α's fresh). Define

$$
\begin{array}{rcll}
pt(E) & = & (B_0^*, \sigma_0^*) & \text{if } \mathrm{Unify}(\mathcal{E}(B_0, E, \sigma_0)) = *, \\
 & = & \mathsf{fail} & \text{if } \mathrm{Unify}(\mathcal{E}(B_0, E, \sigma_0)) = \mathsf{fail}.
\end{array}
$$

The correctness of this procedure follows from Proposition 5.18 and Theorem 5.15. □

Higher-order Typing

The type system can be extended to deal with higher-order functions (see Section 4), using the function space type constructor $\rightarrow$.

The (standard) types of the 'Curried' function symbols $\mathbf{F}_k$ for partial applications are obtained from the (standard) type of $\mathbf{F}$, in the following way (set $\mathbf{F}_{\mathrm{arity}(\mathbf{F})} = \mathbf{F}$).

$$\frac{\mathcal{F}, \mathcal{A} \vdash \mathbf{F}_{k+1} : (\vec{\sigma}, \tau) \rhd \tau'}{\mathcal{F}, \mathcal{A} \vdash \mathbf{F}_k : \vec{\sigma} \rhd (\tau \rightarrow \tau')} \text{(Curry)}$$

The new syntactic construct Ap is treated by an additional rule.

$$\frac{B \vdash E : \sigma \rightarrow \tau \qquad B \vdash E' : \sigma}{B \vdash \mathsf{Ap}(E, E') : \tau} \text{(curried application)}$$

The results of this section (soundness, completeness, subject reduction, principal typing) extend to the higher-order system. As to principal typing, the equations for Ap are generated by

$$\begin{aligned} \mathcal{E}(B, \mathsf{Ap}(E, E'), \sigma) \quad &= \quad \mathcal{E}(B, E, \alpha \rightarrow \sigma) \cup \mathcal{E}(B, E', \alpha) \\ &\qquad \alpha \text{ fresh.} \end{aligned}$$

The function space constructor is allowed inside algebraic type definitions: for example

$$\mathsf{Object}(\alpha) = \mathbf{C}(\alpha, \alpha \rightarrow \alpha, \alpha \rightarrow \mathsf{Int}).$$

6. Introduction to Uniqueness Typing

Uniqueness typing offers the possibility to indicate reference count requirements of functions in the corresponding argument types.

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

Since we deal with both destructive and harmless operations, a purely linear typing system (in which neither copying nor discarding of input is allowed) is too restrictive for our purposes. Instead, we propose to divide the type system into two layers: a 'resource conscious' part in which occurrences are limited, and a 'conventional' part with no reference restrictions. In fact, the former layer (of 'unique' types, indicated by $\bullet$) admits discarding input but excludes copying. Therefore it corresponds to *affine* logic (see Blass (1992)) in which weakening is present but no contraction. The latter layer (of $\times$ types) corresponds to ordinary (intuitionistic) logic with the same strength as conventional typing.

The two layers are connected: it is possible to move from the $\bullet$ layer to the $\times$ layer. These transitions are regulated by the type system: we have a subtype relation allowing a unique object to be seen as a conventional one (in case the accessing function has no reference requirements) and a type correction mechanism (forcing an object with reference count greater than 1 to be regarded as non-unique).

The operator ! in linear logic should not be confused with the attribute $\times$: the type $\sigma^\times$ indicates that there are no *reference* restrictions, whereas $\sigma!$ would stand for 'as many (duplicatewise linear) *copies* as necessary'.

We will now explain the graph-theoretic intuition of our type system. In symbol types, it can be specified that a given argument should be unique (by the annotation $\bullet$). Type correctness means that in any application the concrete function argument should have reference count 1. So the function has indeed 'private' access to its argument, and hence the argument can be updated in-place.

This analysis was intended for inherently destructive operations and their parameters, like **WriteChar**, but can also help to improve storage management. Consider, for instance, the following list reversing function which can be implemented efficiently as a 'destructive' function if the given uniqueness type is used.

$$
\begin{array}{l}
\mathbf{Rev} : \mathsf{List}^\bullet(\alpha^\times) \rhd \mathsf{List}^\bullet(\alpha^\times) \\
\mathbf{Rev}\ (l) \quad \to \quad \mathbf{H}(l, \mathbf{Nil}) \\
\\
\mathbf{H} : (\mathsf{List}^\bullet(\alpha^\times), \mathsf{List}^\bullet(\alpha^\times)) \rhd \mathsf{List}^\bullet(\alpha^\times) \\
\mathbf{H}(\ell_1, \ell_2) \quad \to \quad \mathsf{case}\ \ell_1\ \mathsf{of}\ \ \mathbf{Cons}(h,t) \ \mid\ \mathbf{H}(t, \mathbf{Cons}(h, \ell_2)) \\
\qquad\qquad\qquad\qquad\qquad\quad\ \ \mathbf{Nil} \qquad\ \mid\ \ell_2.
\end{array}
$$

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

We have seen that the environment type $\mathbf{F} : \sigma^\bullet \rhd \cdots$ specifies that **F**'s argument should be unique for **F**. In the same way, uniqueness of results is specified: if $\mathbf{G} : \cdots \rhd \sigma^\bullet$, then a well-typed expression $\mathbf{F}(\mathbf{G}(E))$ remains type-correct, even if $\mathbf{G}(E)$ is subject to computation.

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

From the types given above it can be seen that the layers $\bullet$ and $\times$ are have some internal structure induced by the presence of type constructors: a $\bullet$ type (at the outer-

most level) can have parts marked with $\times$ (and vice versa). This fine structure, with the possibility of specifying affine or conventional behaviour of substructures, is a powerful feature of our system.

Pattern matching (expressed by the case construction) is an essential aspect of term graph rewriting, causing a function to have access to 'deeper' arguments via *data paths* instead of a single reference. This gives rise to 'indirect sharing' of objects by access via intermediate data nodes. For example, if a function **F** has access to a list with non-unique spine, the list elements should also be considered as non-unique for **F**: other functions may access them via the spine. This effect is taken into account by a restriction on the uniqueness types of data constructors: the result of a constructor is unique whenever one of its arguments is. For the constructor **Cons** of lists, for example, the possible uniqueness variant are:

$$\begin{array}{lcll} \textbf{Cons} & : & (\alpha^{\times}, \mathsf{List}^{\times}(\alpha^{\times})) \rhd \mathsf{List}^{\times}(\alpha^{\times}) & (1) \\ \textbf{Cons} & : & (\alpha^{\times}, \mathsf{List}^{\bullet}(\alpha^{\times})) \rhd \mathsf{List}^{\bullet}(\alpha^{\times}) & (2) \\ \textbf{Cons} & : & (\alpha^{\bullet}, \mathsf{List}^{\bullet}(\alpha^{\bullet})) \rhd \mathsf{List}^{\bullet}(\alpha^{\bullet}) & (3) \end{array}$$

With (1) ordinary lists can be built. (2) can be used for lists of which the 'spine' is unique, and (3) for lists of which both the spine and the elements are unique. Observe that in the above example, the List argument of **Cons** is always attributed in the same way as the corresponding List result. In general, such a uniform way of attributing recursive occurrences of a type constructor leads to *homogeneous* data objects: All recursive parts of such an object have the same uniqueness properties as the object itself. A procedure for generating consistent attributions of arbitrary data types can be found in Barendsen and Smetsers (1993).

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

$$\textbf{Cons} : (\alpha^{u}, \mathsf{List}^{v}(\alpha^{u})) \rhd \mathsf{List}^{v}(\alpha^{u})$$

is well-attributed if $v \leq u$. Note that this indeed excludes a constructor for $\mathsf{List}^{\times}(\mathsf{Int}^{\bullet})$. A similar restriction on types of data constructors can be found in the type systems of Guzmán and Hudak (1990) and Turner et al. (1995).

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

The way references are counted can be refined, by making use of information on the evaluation order. To avoid unnecessary complications we will not treat this in detail but give an idea of the method at the end of Section 8.

7. Simple Uniqueness Typing

Algebraic Uniqueness Types

Uniqueness types are constructed from conventional types by assigning a uniqueness attribute to each subexpression. We will denote the attributes as superscripts; for non-variable types these are attached to the topmost type constructor of each type. Below,

$S, T, \ldots$ range over uniqueness types and $u, v, \ldots$ over the attributes $\bullet, \times$. The outermost attribute of S is denoted by $\ulcorner S \urcorner$. Moreover $|S|$ denotes its underlying conventional type.

We will first describe the system without the type constructor $\rightarrow$.

Definition 7.1 The *subtype relation* $\leq$ is very simple: the validity of $S \leq S'$ depends subtypewise on the validity of $u \leq u'$ with u, u' attributes in S, S'. One has, for example,

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

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

Definition 7.2 For each S, we construct the smallest non-unique supertype $[S]$ of S, as follows.

$$\begin{aligned} [\alpha^u] &= \alpha^\times, \\ [\mathsf{T}^u \vec{S}] &= \mathsf{T}^\times \vec{S}. \end{aligned}$$

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

Type correction is applied when dealing with plain sharing (through multiple occurrences of the same variable). In a 'resource conscious' system these variable duplications are easy to detect: they correspond to *contraction* steps in the logical setting. The treatment of recursive objects (occurring as $\mathsf{letrec}\ \vec{x} = \vec{E}\ \mathsf{in}\ E'$) is special: we account for possible cycles by correcting both internal (in $\vec{E}$) and external (in E') references to their roots $\vec{x}$.

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

Definition 7.3 (i) As before, *standard types* of function symbols $(\mathbf{F} : \vec{S} \triangleright T)$ are collected in an environment $\mathcal{F}$.

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

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

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

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

(2) each version is uniqueness propagating;

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

This leaves some freedom as to the choice of attributes on positions not corresponding to $\mathsf{T}, \vec{\alpha}$. Barendsen and Smetsers (1993) offer a general method for constructing a reasonable set of standard types for each constructor. In most cases (like List, see above), however, the choice of attributes of $\mathsf{T}\vec{\alpha}$ fixes those for the $\vec{\sigma_i}$. From now on we assume that

standard types have been determined. For these standard types $\vec{S} \rhd T$ we set $\mathcal{A} \vdash \mathbf{C} : \vec{S} \rhd T$ as before.

Definition 7.4 Symbol types are *instantiated* via the rule

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

Now we are ready to present the derivation system.

Definition 7.5 A *uniqueness typing statement* (in $\mathcal{F}, \mathcal{A}$) has the form

$$B \vdash E : S.$$

Like in linear logic, we have to be precise when dealing with bases used for typing subterms. In particular, duplicating and discarding of inputs are treated explicitly. The denotation B_1, B_2 stands for a disjoint union of bases.

(i) The rules for type assignment are the following.

$$x{:}S \vdash x : S \quad \text{(variable)}$$

$$\frac{\mathcal{F}, \mathcal{A} \vdash \mathbf{S} : \vec{S} \rhd T \qquad B_i \vdash E_i : S_i}{\vec{B} \vdash \mathbf{S}\vec{E} : T} \text{(application)}$$

$$\frac{B \vdash E : S \qquad B', x{:}S \vdash E' : T}{B, B' \vdash \mathsf{let}\ x = E\ \mathsf{in}\ E' : T} \text{(sharing)}$$

$$\frac{B_i, \vec{x}{:}[\vec{S}] \vdash E_i : S_i \qquad B', \vec{x}{:}[\vec{S}] \vdash E' : T}{\vec{B}, B' \vdash \mathsf{letrec}\ \vec{x} = \vec{E}\ \mathsf{in}\ E' : T} \text{(cycle)}$$

$$\frac{B \vdash E : T \qquad \mathcal{F}, \mathcal{A} \vdash \mathbf{C}_i : \vec{S} \rhd T \qquad B', \vec{x_i}{:}\vec{S}_i \vdash E_i : T'}{B, B' \vdash \mathsf{case}\ E\ \mathsf{of}\ \vec{P}|\vec{E} : T' \quad (\text{if } P_i = \mathbf{C}_i \vec{x_i})} \text{(pattern matching)}$$

$$\frac{B \vdash E : S \qquad S \leq S'}{B \vdash E : S'} \text{(subsumption)}$$

Additionally, we have the following 'structural rules'. Weakening expresses that one can discard (unique or non-unique) input. The contraction rule deals with correction of types of shared objects: multiple use of the same object is allowed as long as only non-unique

variants of the types are used.

$$\frac{B \vdash E : T}{B, x{:}S \vdash E : T}\ \text{(weakening)}$$

$$\frac{B, y{:}[S], z{:}[S] \vdash E : T}{B, x{:}S \vdash E[y := x, z := x] : T}\ \text{(contraction)}$$

(ii) Correctness of function definitions is expressed as before: the specification $\mathbf{F}\vec{x} = E$, say with standard type $\vec{S} \rhd T$ for $\mathbf{F}$, is *type correct* if

$$\vec{x}{:}\vec{S} \vdash E : T.$$

A collection of function definitions is type correct if its members are type correct.

In the derivation one clearly recognizes the 'logical' difference between the affine ($\bullet$) and conventional ($\times$) layer: with respect to $\bullet$ one only has weakening as structural rule, whereas $\times$ admits both weakening and contraction since $[S] = S$ if $\ulcorner S \urcorner = \times$.

Higher-Order Uniqueness Types

The treatment of higher-order functions in the uniqueness type system is a subtle matter. Types of partial applications $\mathbf{F}\vec{E}$ need to be assigned a uniqueness attribute. If these partial applications contain unique subexpressions one has to be careful. Consider, for example, a function $\mathbf{F}$ with type $\mathbf{F} : (\sigma^{\bullet}, \tau^{\times}) \rhd \sigma^{\bullet}$ in the partial application $\mathbf{F}E$. Clearly, the result type of this application is of the form $\tau^{\times} \xrightarrow{u} \sigma^{\bullet}$. If one allows that this application is used more than once, one cannot guarantee that the argument E (with type $\sigma^{\bullet}$) remains unique during evaluation. E.g. if $\mathbf{F}E$ is passed to a function $\mathbf{G}(f) = (f\mathbf{0}, f\mathbf{1})$, the occurrences of f will result in two applications of $\mathbf{F}$ sharing the same expression E. Apparently, the $\mathbf{F}E$ expression is *necessarily* unique: its reference count should never become greater than 1, i.e. it is not allowed to move from the affine to the conventional type layer. There are several ways to achieve this. For instance, one might introduce a new uniqueness attribute, say $\triangle$, for any unique object that does not coerce to a non-unique variant. This has been described in Barendsen and Smetsers (1993). Another solution is the region-administration introduced by Reynolds (1995).

Instead of introducing a new attribute, the present paper assigns the attribute $\bullet$ to the above $\rightarrow$-type, but considers the $\rightarrow$ constructor in combination with the $\bullet$ attribute as special: it is not permitted to discard its uniqueness. The leads to an adjustment of the subtyping relations as well as of the type correction operator $[\cdot]$.

As to the subtyping relation, the attributes of corresponding occurrences of the $\rightarrow$ constructors (in the left-hand and the right-hand side of an inequality) should be identical. The same is required (to ensure substitutivity of the subtyping relation) for variables.

The subtyping relation becomes inherently more complex than in the algebraic case because of the so-called *contravariance* of $\rightarrow$ in its first argument:

$$S \xrightarrow{u} S' \leq T \xrightarrow{u} T' \quad \Leftrightarrow \quad T \leq S,\ S' \leq T'.$$

Since $\to$ may appear in the definitions of algebraic type constructors, these constructors may inherit the co- or contravariant subtyping behaviour with respect to their arguments. We can classify the 'sign' of the arguments of each type constructor as $\oplus$ (positive, covariant), $\ominus$ (negative, contravariant) or $\top$ (both positive and negative). In general this is done by analyzing the (possibly mutually recursive) algebraic type definitions by a fixedpoint construction, with basis $\mathit{sign}(\to) = (\ominus, \oplus)$.

Notation. The variants $\leq^{\oplus}$, $\leq^{\ominus}$ and $\leq^{\top}$ are defined in terms of $\leq$, as follows.

$$\begin{aligned} S \leq^{\oplus} T &\quad\Leftrightarrow\quad S \leq T, \\ S \leq^{\ominus} T &\quad\Leftrightarrow\quad T \leq S, \\ S \leq^{\top} T &\quad\Leftrightarrow\quad S \leq^{\oplus} T \text{ and } S \leq^{\ominus} T. \end{aligned}$$

Moreover we set

$$\vec{S} \leq^{\vec{s}} \vec{T} \;\Leftrightarrow\; S_i \leq^{s_i} T_i \quad \text{for each } i.$$

Definition 7.6 The subtyping relation $\leq$ is defined by induction.

$$\begin{aligned} \alpha^u \leq \alpha^v &\quad\Leftrightarrow\quad u = v, \\ \mathsf{T}^u \vec{S} \leq \mathsf{T}^v \vec{T} &\quad\Leftrightarrow\quad u \leq v \text{ and } \vec{S} \leq^{\mathit{sign}(\mathsf{T})} \vec{T}, \\ S \xrightarrow{u} S' \leq T \xrightarrow{v} T' &\quad\Leftrightarrow\quad u = v \text{ and } S \leq^{\ominus} T \text{ and } S' \leq^{\oplus} T'. \end{aligned}$$

Then we have, for example,

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

Adjusting the type correction operator is easy: correction of $\xrightarrow{\bullet}$ types simply fails. Thus the operator $[\cdot]$ becomes a partial function:

$$\begin{aligned} [\alpha^u] &= \alpha^{\times} && \text{if } u = \times, \\ [\mathsf{T}^u \vec{S}] &= \mathsf{T}^{\times} \vec{S}, && \\ [S \xrightarrow{u} T] &= S \xrightarrow{\times} T && \text{if } u = \times, \\ [S] &= \uparrow \text{(undefined)} && \text{in all other cases.} \end{aligned}$$

The type of $\mathbf{F}_k$ is defined in terms the type of $\mathbf{F}_{k+1}$ by the following rule.

$$\frac{\mathcal{F}, \mathcal{A} \vdash \mathbf{F}_{k+1} : (\vec{S}, T) \rhd T' \qquad u \leq \Pi^{\ulcorner} \vec{S}^{\urcorner}}{\mathcal{F}, \mathcal{A} \vdash \mathbf{F}_k : \vec{S} \rhd (T \xrightarrow{u} T')} \text{(Curry)}$$

Here, $\Pi\vec{u}$ stands for the so-called *cumulative* uniqueness attribute of $\vec{u}$: it equals $\bullet$ whenever some u_i is $\bullet$, and $\times$ otherwise.

The typing rule for Ap is defined straightforwardly.

$$\frac{B \vdash E : S \xrightarrow{u} T \qquad B' \vdash E' : S}{B, B' \vdash \mathsf{Ap}(E, E') : T} \text{(curried application)}$$

Semantics

We start with defining a notion of uniqueness typing for graphs, based on type assignment to nodes in graphs. Subtyping and type correction will be done along references to objects, whereas on expressions one can perform coercions regardless of their context. This difference becomes apparent in the root of a graph. By making a small adjustment (introduced in Barendsen and Smetsers (1993) as a tool to obtain subject reduction) we can reconcile the two approaches.

Definition 7.7 (i) Let g be a graph. Then g^+ is the graph that results from g by adding a new root r^+ with in-degree 0 containing the data symbol **Root** of arity 1, pointing to the root r of g.

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

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

Definition 7.8 Let n be a node in g. The *reference count* of n in g (notation $\mathrm{rc}_g(n)$ or just $\mathrm{rc}(n)$) is $\otimes$ if n appears more than once in the right-hand sides of equations in g^+, and $\odot$ otherwise.

Note that the above mechanism only differs from ordinary reference counting at the root of the graph, notably when the root is part of a cycle.

Definition 7.9 (TGRS uniqueness graph typing) Let g be a graph.

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

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

and for all $i \leq k$

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

(ii) We say that $\mathcal{T}$ *types* g with S (notation $\mathcal{T}(g) = S$) if moreover $\mathcal{T}(r^+) = S$. Furthermore g is *typable* with S (notation $g : S$) if $\mathcal{T}(g) = S$ for some $\mathcal{T}$.

The constraints in the above definition reflect the uniqueness property of function applications mentioned in Section 6. If, say, **F** with arity 2 has a standard type in which the first argument is unique, then for any application $x = \mathbf{F}(y, z)$ in a type correct graph g we have that $\mathrm{rc}_g(y) = 1$. The following subsection shows that this property is indeed established by the natural deduction system.

Soundness

As a first step towards the soundness proof we need the following technical results.

Definition 7.10 (i) $[S]^{\otimes} = [S], [S]^{\odot} = S$.
(ii) By $\mathcal{T} \models B$ we denote that $\mathcal{T}(x) = S$ for each $(x{:}S) \in B$.

Lemma 7.11 *Let $B \vdash E : S$. Set $r_E = r_{[\![E]\!]}$, $\mathrm{rc}_E = \mathrm{rc}_{[\![E]\!]}(r_E)$. Then there exists a type assignment $\mathcal{T}$ such that $\mathcal{T}$ is a uniqueness typing for $[\![E]\!]$ and moreover*

$$\mathcal{T} \models B,$$
$$[\mathcal{T}(r_E)]^{\mathrm{rc}_E} \leq S.$$

Proof. By induction on the derivation of $B \vdash E : S$. We will only consider two cases: *application* and *contraction*. All other cases are handled in the same way.

• $\vec{B} \vdash \mathbf{S}\vec{E} : T$ since $\mathcal{F}, \mathcal{A} \vdash \mathbf{S} : \vec{S} \rhd T$ and $B_i \vdash E_i : S_i$. By induction hypothesis we have $\mathcal{T}_i$ for $[\![E_i]\!]$ with $\mathcal{T}_i \models B_i$ and $[\mathcal{T}(r_{E_i})]^{\mathrm{rc}_{E_i}} \leq S_i$. Note that $\mathrm{rc}_{E_i} = \mathrm{rc}_{[\![\mathbf{S}\vec{E}]\!]}(r_{E_i})$. Set $\mathcal{T} = \bigcup_i \mathcal{T}_i \cup \{r_{\mathbf{S}\vec{E}}, r^+ \mapsto T\}$. Then $\mathcal{T}$ satisfies the requirements.

• $B, x{:}S \vdash E[y, z := x] : T$ since $B, y{:}[S], z{:}[S] \vdash E : T$. By induction hypothesis we have $\mathcal{T} \models B, y{:}[S], z{:}[S]$ and $[\mathcal{T}(r_E)]^{\mathrm{rc}_E} \leq T$. Set $\mathcal{T}' = \mathcal{T} \cup [x \mapsto S]$. Then $\mathcal{T}' \models B, x{:}S$ and $\mathcal{T}'$ is a uniqueness typing for $[\![E[y, z := x]]\!] = [\![E]\!][y, z := x]$ by transitivity of $\leq$, using $S \leq [S]$ and $[S] \leq [S]$ and we are done. □

Definition 7.12 Let E be an expression.
(i) We say that $\mathcal{T}$ *types E with S* (notation $\mathcal{T} \models E : S$) if $\mathcal{T}$ is a uniqueness typing for $[\![E]\!]$ such that

$$\mathcal{T}([\![E]\!]) = S.$$

(ii) By $B \models E : S$ we denote that B is extendible to a typing of $[\![E]\!]$, i.e., for some $\mathcal{T}$ one has

$$\mathcal{T} \models B, \quad \mathcal{T} \models E : S.$$

Theorem 7.13 (Soundness of expression uniqueness typing) *For any B, E, S*

$$B \vdash E : S \;\Rightarrow\; B \models E : S.$$

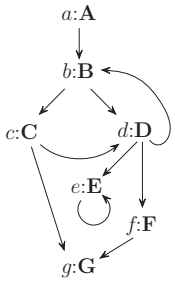
Proof. Suppose $B \vdash E : S$. By Lemma 7.11 there exists a type assignment $\mathcal{T}$ for $[\![E]\!]$ with $[\mathcal{T}(r_E)]^{\mathrm{rc}_E} \leq S$. Set $\mathcal{T}' = \mathcal{T}[r_E^+ \mapsto S]$. Then $\mathcal{T}'$ is a uniqueness typing for $[\![E]\!]$. (The additional root reference allows an extra 'top coercion'.) Moreover $\mathcal{T}' \models B$ and $\mathcal{T}' \models E : S$. □

Completeness

Showing that the inductive uniqueness type system is powerful enough to capture uniqueness typing of graphs is more involved than for the conventional case. This is because our typing rules are rather *intensional*: Due to the pessimistic treatment of letrec-expressions

(type correction even in the case of a degenerate cycle) there are expressions E, E' and type S such that $[\![E]\!] = [\![E']\!]$, E is typable with S whereas E' is not typable at all.

The idea is to transform a given graph $g = \langle r \,|\, G \rangle$ into an expression $g^{\circledast}$ that contains a 'minimal' amount of letrec-expressions. This is done by stepwise substitution of equations in G corresponding to nodes with reference count 1. Take, for example, the following graph.



The intended procedure leads to the expression

$$\begin{array}{lllll} \mathsf{letrec} & b & = & \mathbf{B}(\mathbf{C}(g,d),d), \\ & d & = & \mathbf{D}(e,\mathbf{F}(g),b), \\ & e & = & \mathbf{E}(e), \\ & g & = & \mathbf{G} \\ \mathsf{in} & & & \mathbf{A}(b) \end{array}$$

Definition 7.14 Let $g = \langle r \,|\, G \rangle$ be a graph. Say E_n is the right-hand side of the equation in G corresponding to variable n.

(i) The expressions E_n are translated into expressions ${E_n}^*$, as follows.

$$(\mathbf{S}(n_1, \ldots, n_k))^* = \mathbf{S}(\overline{n_1}, \ldots, \overline{n_k}).$$

Moreover

$$\begin{array}{rcll} \overline{n} & = & n & \text{if } \mathrm{rc}_g(n) = \otimes \text{ or } n \in \mathrm{FV}(g) \\ & = & {E_n}^* & \text{otherwise.} \end{array}$$

(ii) Finally, the *standard* expression denoting g (notation $g^{\circledast}$) is given by

$$g^{\circledast} = \mathsf{letrec}\ \vec{n} = \vec{E_n}^*\ \mathsf{in}\ \overline{r},$$

where $\vec{n}$ is the collection of bound variables with reference count $\otimes$.

The above transformation is an instance of the *hiding* operation defined by Ariola and Klop (1995).

It is not completely trivial that the (mutually recursive) definition of * and $\bar{\cdot}$ is sound. To see that this is indeed the case consider the following measure on g-nodes.

Definition 7.15 Let $n \in V(g)$. Then n is said to have *degree* 0 if $n \in \mathrm{FV}(g)$ or n has no arguments with reference count $\odot$. Otherwise, if the maximal degree of n's arguments with reference count $\odot$ is d, then n has degree $d + 1$.

This is a sound definition: since there exists no infinite path of nodes with reference count $\odot$, each node n can be assigned a unique, finite degree, denoted by $\deg(n)$.

It is clear that the degree of nodes decreases with each recursive occurrence of the operator * above.

Lemma 7.16 $[\![g^{\circledast}]\!] = g$

Proof. Obvious. □

Below we will use the degree as a technical tool to prove that $g^{\circledast}$ can be typed with the same type as g. In the sequel, fix a graph g and an uniqueness typing $\mathcal{T}$ for g.

Definition 7.17 (i) The *initial basis* (of g) (notation B^{I}) is the set

$$B^{\mathrm{I}} = \{n : [\mathcal{T}(n)]^{\mathrm{rc}_g(n)} \mid n \in \mathrm{FV}(g)\}.$$

(ii) The *recursion basis* (notation B^{R}) is the set

$$B^{\mathrm{R}} = \{n : [\mathcal{T}(n)] \mid \mathrm{rc}_g(n) = \otimes\}.$$

Lemma 7.18 *For all* $n \in \mathrm{V}(g)$ *one has*
(i) $B^{\mathrm{I}}, B^{\mathrm{R}} \vdash {E_n}^* : \mathcal{T}(n)$;
(ii) $B^{\mathrm{I}}, B^{\mathrm{R}} \vdash \overline{n} : [\mathcal{T}(n)]^{\mathrm{rc}_g(n)}$.

Proof. By simultaneous course-of-values induction on the degree of n. Suppose (i) and (ii) hold for all nodes with degree $< d$. Let n have degree d.

(i) Say ${E_n}^* = \mathbf{S}(\overline{n_1}, \ldots, \overline{n_k})$ and $\mathcal{F}, \mathcal{A} \vdash \mathbf{S} : \vec{S} \rhd T$ with $\mathcal{T}(n) = T$ and $[\mathcal{T}(n_i)]^{\mathrm{rc}(n_i)} \leq S_i$ for all $i \leq k$.

Claim. $B^{\mathrm{I}}, B^{\mathrm{R}} \vdash \overline{n} : S_i$. Then we are done: we can complete the derivation using the rules *application* and *contraction*. The latter deals with multiple occurrences of the variables in $B^{\mathrm{I}}, B^{\mathrm{R}}$; note that $[\cdot]$ is idempotent, i.e. $[[S]] = [S]$ for each type S.

Proof. Let $i \leq k$.

Case 1. $\mathrm{rc}(n_i) = \odot$ and $n_i \in \mathrm{BV}(g)$. Then by induction hypothesis (ii) (note that $\deg(n_i) < d$) we have

$$B^{\mathrm{I}}, B^{\mathrm{R}} \vdash \overline{n_i} : \mathcal{T}(n_i).$$

Now we can apply *subsumption*.

Case 2. $\mathrm{rc}(n_i) = \otimes$ and $n_i \in \mathrm{BV}(g)$. Then

$$B^{\mathrm{I}}, B^{\mathrm{R}} \vdash \overline{n_i} : [\mathcal{T}(n_i)],$$

by *variable* and *weakening*, so by *subsumption* the result follows.

Case 3. $n_i \in \mathrm{FV}(g)$. Then $B^{\mathrm{I}} \vdash n_i : [\mathcal{T}(n_i)]^{rc(n_i)}$. Again by *subsumption* the result follows. $\square_{Claim}$

(ii) The case $\mathrm{rc}(n) = \odot$ and $n \in \mathrm{BV}(g)$ is covered by (i). Otherwise, either B^{R} (in case $n \in \mathrm{BV}(g)$) or B^{I} (in case $n \in \mathrm{FV}(g)$) contains $n : [\mathcal{T}(n)]^{rc(n)}$. Now we are done by *variable* and *weakening*. □

Proposition 7.19 $B^{\mathrm{I}} \vdash g^{\circledast} : \mathcal{T}(g)$.

Proof. Since $\mathcal{T}$ is a typing for g we have $[\mathcal{T}(r)]^{\mathrm{rc}(r)} \leq \mathcal{T}(r^+)$ by the standard type for **Root**. Now by Lemma 7.18, using the rule *cycle* and the rule *weakening* (also to deal with multiple occurrences of the variables in B^{I}) and *subsumption* the result follows. □

Theorem 7.20 (Completeness for expression uniqueness typing)

$$B \models E : S \qquad \Rightarrow \qquad \exists E' \sim E \; [B \vdash E' : S].$$

Proof. Without loss of generality, we can assume that B contains no declarations for variables not appearing in E. Suppose $B \models E : S$; say $\mathcal{T} \models B$ and $\mathcal{T}([\![E]\!]) = S$. Observe that $B = B^{\mathrm{I}}$. Set $E' = [\![E]\!]^{\circledast}$. Then $E' \sim E$ by Lemma 7.16 and we are done by Proposition 7.19. □

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

Definition 7.21 (TGRS Rule uniqueness typing)
(i) A uniqueness type assignment $\mathcal{T}$ to variables can be extended to patterns in the following way:

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

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

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

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

Proposition 7.22 (Soundness of uniqueness function typing)

$$\mathbb{F} \textit{ is uniqueness type correct } \Rightarrow\; [\![\mathbb{F}]\!] \textit{ is uniqueness type correct.}$$

Proof. Straightforward. □

Subject Reduction

Theorem 7.23 (TGRS Uniqueness reduction typing) *Suppose $\mathcal{R}$ is uniqueness type correct. Then for any g, h, S*

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

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

Proof. See Barendsen and Smetsers (1993). □

Finally, by combining this result with the results from the previous subsection we can formulate the soundness of the uniqueness type system for expressions with respect to graph rewriting.

Subject Reduction Theorem 7.24 *Suppose* $\mathbb{F}$ *is type correct. Then*

$$\left.\begin{array}{l} B \vdash E : S \\ E \twoheadrightarrow_{\mathbb{F}} E' \end{array}\right\} \quad \Rightarrow \quad \exists E'' \sim E' \, [B \vdash E'' : S].$$

Proof. First note that $[\![\mathbb{F}]\!]$ is type correct by Proposition 7.22. Suppose $B \vdash E : S$ and $E \twoheadrightarrow_{\mathbb{F}} E'$. Then $B \models E : S$ by soundness (Theorem 7.13). By Theorem 7.23 we have $B \models E' : S$. Hence by completeness (Theorem 7.20) the result follows. □

8. Polymorphic Uniqueness Typing

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

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

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

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

Syntax

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

Definition 8.1 (i) As to the attribute relation $\leq$, we say that $u \leq v$ is *derivable* from the attribute environment Γ (notation $\Gamma \vdash u \leq v$) if $\Gamma \vdash u \leq v$ can be produced by the axioms

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

and rule

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

(ii) This denotation is extended to finite sets of inequalities: $\Gamma \vdash \Gamma'$ if $\Gamma \vdash u \leq v$ for each $(u \leq v) \in \Gamma'$. By $u = v$ we denote the pair $u \leq v, v \leq u$.

(iii) We say that Γ is *consistent* if $\Gamma \nvdash \times \leq \bullet$.

Definition 8.2 (i) For every Γ, the subtyping relation $\leq_\Gamma$ is defined by induction.

$$\alpha^u \leq_\Gamma \alpha^v \quad \Leftrightarrow \quad \Gamma \vdash u = v,$$

$$\begin{aligned} \mathsf{T}^u\vec{S} \le_\Gamma \mathsf{T}^v\vec{T} \quad &\Leftrightarrow \quad \Gamma \vdash u \le v \text{ and } \vec{S} \le_\Gamma^{sign(\mathsf{T})} \vec{T}, \\ S \xrightarrow{u} S' \le_\Gamma T \xrightarrow{v} T' \quad &\Leftrightarrow \quad \Gamma \vdash u = v \text{ and } S \le_\Gamma^{\ominus} T \text{ and } S' \le_\Gamma^{\oplus} T'. \end{aligned}$$

(ii) The types S and T are *equal modulo* Γ (notation $S =_\Gamma T$) if $S \le_\Gamma T$ and $T \le_\Gamma S$.

One has, for example,

$$\mathsf{List}^u(\mathsf{Int}^v \xrightarrow{w} \alpha^x) \le_\Gamma \mathsf{List}^{u'}(\mathsf{Int}^{v'} \xrightarrow{w'} \alpha^{x'}) \quad \text{iff} \quad \Gamma \vdash u \le u', v' \le v, w = w', x = x'.$$

Definition 8.3 The context rules for the polymorphic system are

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

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

$$\frac{\mathcal{F}, \mathcal{A} \vdash \mathbf{F}_{i+1} : (\vec{S}, T) \rhd T' \mid \Gamma \qquad \Gamma \vdash u \le \ulcorner \vec{S} \urcorner}{\mathcal{F}, \mathcal{A} \vdash \mathbf{F}_i : \vec{S} \rhd (T \xrightarrow{u} T') \mid \Gamma} \text{(Curry)}$$

The correction operation is also relativized.

Definition 8.4 For every Γ, the Γ-*correction* of types is the partial function $[\cdot]_\Gamma$ defined inductively as follows.

$$\begin{aligned} [\alpha^u]_\Gamma &= \alpha^u && \text{if } \Gamma \vdash u = \times, \\ [\mathsf{T}^u\vec{S}]_\Gamma &= \mathsf{T}^\times\vec{S}, && \\ [S \xrightarrow{u} T]_\Gamma &= S \xrightarrow{u} T && \text{if } \Gamma \vdash u = \times, \\ [S]_\Gamma &= \uparrow && \text{otherwise.} \end{aligned}$$

The rules for producing typing statements $B \vdash_\Gamma E : S$ are obtained from the previous ones, roughly by replacing $\vdash$ by $\vdash_\Gamma$. The application rule, for example, becomes

$$\frac{\mathcal{F}, \mathcal{A} \vdash \mathbf{S} : \vec{S} \rhd T \mid \Gamma \qquad B_i \vdash_\Gamma E_i : S_i}{\vec{B} \vdash_\Gamma \mathbf{S}\vec{E} : T} \text{(application)}$$

and the subsumption rule

$$\frac{B \vdash_\Gamma E : S \qquad S \le_\Gamma S'}{B \vdash_\Gamma E : S'} \text{(subsumption)}$$

The environments Γ in the deduction system are *global* in the sense that they may contain auxiliary uniqueness constraints (attribute inequalities appearing in some derivation

step, but not occurring in the final basis and type). In order to eliminate these superfluous constraints in the conclusion of a deduction, we refine the notion of derivability.

Definition 8.5 (i) Let Γ, Γ' be coercion environments, and let S be a uniqueness type. Then Γ, Γ' are *equivalent* with respect to S (notation $\Gamma \sim_S \Gamma'$) if for all attributes u, v in S, $\bullet, \times$

$$\Gamma \vdash u \leq v \;\Leftrightarrow\; \Gamma' \vdash u \leq v.$$

(ii) This denotation is extended to bases and/or types: we write $\Gamma \sim_{B,S} \Gamma'$ if $\Gamma \sim_T \Gamma'$ for all T appearing in B, S.

Definition 8.6 (i) A *polymorphic uniqueness typing statement* is an expression of the form

$$B \vdash E : S \mid \Gamma.$$

Such a statement is *derivable* if there exists a consistent Γ' such that $\Gamma' \sim_{B,S} \Gamma$ and $B \vdash_{\Gamma'} E : S$ can be produced via the above axioms and rules.

(ii) Say the standard type for $\mathbf{F}$ in $\mathcal{F}$ is $\vec{S} \rhd T \mid \Gamma$. Then the function definition $\mathbf{F}\vec{x} = E$ is *type correct* if

$$\vec{x}{:}\vec{S} \vdash E : T \mid \Gamma.$$

Formulating the notion of attribute instance (via attribute substitutions $\diamond$) for polymorphic uniqueness typing is more subtle than for conventional type instantiation. One has to take into account that the uniqueness information is divided into two parts: uniqueness types and coercion environments. For example, the typings $\mathsf{Int}^a \mid a \leq \bullet$ and $\mathsf{Int}^\bullet$ have the same 'uniqueness content', but the uniqueness information is denoted in two different ways. The following definition of instantiation focusses on the uniqueness content, abstracting from the specific type denotation.

Definition 8.7 (i) The typing $S' \mid \Gamma'$ is a *uniqueness instance* of $S \mid \Gamma$ (notation $S'|\Gamma' \subseteq S|\Gamma$) if S, S' have the same conventional skeleton ($|S| = |S'|$) and there exists an attribute substitution $\diamond$ such that

$$\Gamma' \vdash \Gamma^\diamond, \qquad S' =_{\Gamma'} S^\diamond.$$

(ii) We also use this denotation for sequences of types: $\vec{S'}|\Gamma' \subseteq \vec{S}|\Gamma$.

The following expresses that attribute variables and attribute environments can be regarded as uniqueness schemes: all instances of a given typing are correct.

Proposition 8.8

$$\left.\begin{array}{r} B \vdash E : S \mid \Gamma \\ B',S'|\Gamma' \subseteq B,S|\Gamma \end{array}\right\} \;\Rightarrow\; B' \vdash E : S' \mid \Gamma'.$$

Proof. Induction on the derivation showing $B \vdash E : S \mid \Gamma$. □

Semantics

The semantic results of Section 7 go through in the polymorphic system. Below we reformulate the main results.

Definition 8.9 (TGRS polymorphic uniqueness graph typing) Let $g = \langle r \mid G \rangle$ be a graph. A *polymorphic uniqueness typing* for g is a pair $\langle \mathcal{T}, \Gamma \rangle$ where $\mathcal{T}$ is a uniqueness type assignment to nodes in g and Γ is a consistent coercion environment such that for any equation $x = \mathbf{S}(\vec{y})$ in G there exist types $\vec{S}$ with the following properties.

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

and for all $i \leq k$

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

The following definition prepares for the soundness result.

Definition 8.10 E an expression and S a uniqueness type.

(i) Let B be a basis. By $\mathcal{T} \models_\Gamma B$ we denote that $\mathcal{T}(x) =_\Gamma T$ for each $(x{:}T)$ in B.

(ii) We say that $\mathcal{T}$ types E with S in Γ (notation $\mathcal{T} \models_\Gamma E : S$) if $\langle \mathcal{T}, \Gamma \rangle$ is a uniqueness typing for $[\![E]\!]^+$ such that $\mathcal{T}([\![E]\!]^+) =_\Gamma S$.

(iii) By $B \models E : S \mid \Gamma$ we denote that for some $\mathcal{T}$ and $\Gamma' \sim_{B,S} \Gamma$

$$\mathcal{T} \models_{\Gamma'} B, \quad \mathcal{T} \models_{\Gamma'} E : S.$$

Theorem 8.11 (Soundness) $B \vdash E : S \mid \Gamma \;\Rightarrow\; B \models E : S \mid \Gamma$.

Theorem 8.12 (Completeness) $B \models E : S \mid \Gamma \;\Rightarrow\; \exists E' \sim E\, [B \vdash E' : S \mid \Gamma]$.

Subject Reduction Theorem 8.13 *Suppose* $\mathbb{F}$ *is type correct. Then*

$$\left.\begin{array}{r} B \vdash E : S \mid \Gamma \\ E \twoheadrightarrow_{\mathbb{F}} E' \end{array}\right\} \quad \Rightarrow \quad \exists E'' \sim E'\, [B \vdash E'' : S \mid \Gamma].$$

Uniqueness Type Inference

In this section we will describe how to compute uniqueness variants of conventional typings. The presentation will proceed along the same lines as the conventional case: given an expression, we collect a 'minimal' set of requirements (this time in the form of inequalities). It will be decidable whether this set has a solution; moreover, in the positive case a 'principal' solution is computable. For the graph theoretic setting, uniqueness type inference has been described in Barendsen and Smetsers (1995b).

In order to generate uniqueness requirements in an inductive way, we construct a syntax-directed variant of the type derivation system. The substitution in the contraction rule is a global operation which does not fit into a syntax-directed (decompositional)

system. Therefore we introduce *variable annotations* for administration of multiple variable occurrences.

We assume that the applied occurrences of variables in our expressions are marked: $x^{\otimes}$ if x either occurs more than once or x is a letrec-variable, and $x^{\odot}$ if x occurs only once. Defining occurrences (let $x = \cdots$) are not marked. This marking corresponds to a simple reference-count determination in graphs.

The subsumption rule is incorporated in the rules (variable) and (application). The contraction rule has become obsolete by our new administration of sharing: uniqueness correction can now be combined with the (variable) rule. Weakening is taken into account by allowing a larger basis in the rules. Moreover, strict separation of bases for typing is not longer necessary, again by our local administration of multiple variable occurrences.

Definition 8.14 The syntax directed polymorphic system looks as follows.

$$\frac{S \leq_\Gamma S'}{B, x{:}S \ \vdash_\Gamma \ x^{\odot} : S'} \text{(variable } \odot\text{)} \qquad \frac{[S]_\Gamma \leq_\Gamma S'}{B, x{:}S \ \vdash_\Gamma \ x^{\otimes} : S'} \text{(variable } \otimes\text{)}$$

$$\frac{\mathcal{F}, \mathcal{A} \vdash \mathbf{S} : \vec{S} \rhd T \mid \Gamma \qquad B \ \vdash_\Gamma \ E_i : S_i \qquad T \leq_\Gamma T'}{B \ \vdash_\Gamma \ \mathbf{S}\vec{E} : T'} \text{(application)}$$

$$\frac{B \ \vdash_\Gamma \ E : S \qquad B, x{:}S \ \vdash_\Gamma \ E' : T}{B \ \vdash_\Gamma \ \mathsf{let}\ x = E\ \mathsf{in}\ E' : T} \text{(sharing)}$$

$$\frac{B, \vec{x}{:}\vec{S} \ \vdash_\Gamma \ \vec{E} : \vec{S} \qquad B, \vec{x}{:}\vec{S} \ \vdash_\Gamma \ E' : T}{B \ \vdash_\Gamma \ \mathsf{letrec}\ \vec{x} = \vec{E}\ \mathsf{in}\ E' : T} \text{(cycle)}$$

$$\frac{B \ \vdash_\Gamma \ E : T \qquad \mathcal{F}, \mathcal{A} \ \vdash_\Gamma \ \mathbf{C}_i : \vec{S} \rhd T \qquad B, \vec{x_i}{:}\vec{S_i} \ \vdash_\Gamma \ E_i : T'}{B \ \vdash_\Gamma \ \mathsf{case}\ E\ \mathsf{of}\ \vec{P}|\vec{E} : T' \quad (\text{if } P_i = \mathbf{C}_i\vec{x_i})} \text{(pattern matching)}$$

Derivability in the resulting (syntax directed) system is denoted by $\vdash^{\mathrm{sd}}$.

Theorem 8.15 *If E is well-marked, then*

$$B \ \vdash^{\mathrm{sd}}_\Gamma \ E : S \ \Leftrightarrow \ B \ \vdash_\Gamma \ E^- : S,$$

where E^- results from E by removing all markings $\odot$, $\otimes$.

We need an equivalent of the statements $\mathcal{E}$ that were used in the conventional case.

Definition 8.16 (i) An *attribute inequality* is an expression of the form $u \leq v$, where u, v are uniqueness attributes. A *type inequality statement* is an expression of the form $S \preceq T$, where S, T are uniqueness types. The expression $S \simeq T$ stands for the combination $S \preceq T, T \preceq S$.

(ii) A system of *uniqueness requirements* is a pair $\mathcal{U} = \langle \mathcal{S}, \Delta \rangle$, where $\mathcal{S}$ consists of type inequality statements and Δ of attribute inequality statements.

(iii) For each $\mathcal{U}$, the set $|\mathcal{U}|$ is the underlying conventional set of equations $|S| = |T|$ associated with the inequality statements $S \preceq T$ in $\mathcal{U}$.

Definition 8.17 (i) A *uniqueness type substitution* is an assignment $*$ of uniqueness types to uniqueness type variables such that $\ulcorner *(\alpha^u) \urcorner = u$ for each α^u in $*$'s domain.

(ii) A *solution* for a system $\mathcal{U} = \langle \mathcal{S}, \Delta \rangle$ consists of a substitution $*$ and a consistent coercion environment Γ such that

(1) $S^* \leq_\Gamma T^*$ for each $(S \preceq T)$ in $\mathcal{S}$.

(2) $\Gamma \vdash \Delta$.

In this case we write $*, \Gamma \models \mathcal{U}$.

In the sequel we will express uniqueness typing constraints in the form of a system $\mathcal{U}$. Towards a procedure for solving such a system, we consider the case where a partial solution $*$ has been determined that 'conventionally satisfies' $\mathcal{U}$, i.e. $|*| \models |\mathcal{U}|$. We now focus on determining a suitable Γ which solves the remaining inequalities by so-called *uniqueness unification*.

In view of this, the unification algorithm takes 'conventionally correct' systems as input, i.e., systems in which for all inequality statements $S \preceq T$ we have $|S| = |T|$.

Lemma 8.18 *Consistency of attribute environments is a decidable property.*

Proof. Note that Γ is consistent iff the 'transitive closure' of Γ does not contain $\times \leq \bullet$. This can be verified effectively by the finiteness of attribute environments. □

Proposition 8.19 (Uniqueness Unification) *There exists a computable function attr, having as input conventionally correct systems of uniqueness requirements and returning an attribute environment or* fail*, such that*

$$\begin{array}{rcl} attr(\mathcal{U}) = \Gamma & \Rightarrow & \Gamma \models \mathcal{U} \text{ and } \Gamma \text{ is consistent,} \\ & & \text{and } \Gamma' \vdash \Gamma \text{ for each consistent } \Gamma' \models \mathcal{U} \\ attr(\mathcal{U}) = \mathsf{fail} & \Rightarrow & \text{there is no consistent } \Gamma \text{ such that } \Gamma \models \mathcal{U}. \end{array}$$

Proof. For each S, T with $|S| = |T|$, the set $\Gamma(S,T)$ (ensuring $S \leq T$) is defined inductively as follows.

Notation. $\Gamma^{\oplus}(S,T) = \Gamma(S,T), \quad \Gamma^{\ominus}(S,T) = \Gamma(T,S),$
$\Gamma^{\top}(S,T) = \Gamma^{\oplus}(S,T) \cup \Gamma^{\ominus}(S,T),$
$\Gamma^{\vec{s}}(S,T) = \bigcup_i \Gamma^{s_i}(S_i, T_i).$

Now set

$$\begin{array}{rcl} \Gamma(\alpha^u, \alpha^v) & = & \{u = v\}, \\ \Gamma(S \xrightarrow{u} S', T \xrightarrow{v} T') & = & \Gamma^{\ominus}(S,T) \cup \Gamma^{\oplus}(S',T') \cup \{u = v\}, \\ \Gamma(\mathsf{T}^u(\vec{S}), \mathsf{T}^v(\vec{T})) & = & \Gamma^{sign(\mathsf{T})}(\vec{S}, \vec{T}) \cup \{u \leq v\}. \end{array}$$

Say $\mathcal{U} = \langle \mathcal{S}, \Delta \rangle$. Finally, set

$$\Gamma = \bigcup_{`S \preceq T' \in \mathcal{S}} \Gamma(S,T) \cup \Delta.$$

Clearly, Γ is computable from $\mathcal{U}$. If Γ is consistent (Lemma 8.18), output $attr(\mathcal{U}) = \Gamma$; otherwise $attr(\mathcal{U}) = \mathsf{fail}$. □

In practice, the strategy to solve a system $\mathcal{U}$ will be based on a conventional typing algorithm. We will describe how to combine the attribution procedure with a lifting concept: given a system $\mathcal{U}$ and a *conventional* solution $*_0$ of $|\mathcal{U}|$ we can compute a 'most general' uniqueness variant $*, \Gamma$ of $*_0$ (if it exists) solving $\mathcal{U}$. The result is called an *attribution* of $*_0$.

We first have to define the notion 'most general attribution', using the concept of attribute instantiation, cf. Definition 8.7.

Definition 8.20 (i) The pair $(*', \Gamma')$ is an instance of $(*, \Gamma)$ if for some $\diamond$

$$\Gamma' \vdash \Gamma^\diamond, \qquad *' =_{\Gamma'} *^\diamond,$$

where the latter equation is to be taken argumentwise.

(ii) A (conventional) solution $*_0$ of $|\mathcal{U}|$ is called *attributable* if there exists a uniqueness substitution $*$ with $|*| = *_0$ and a consistent environment Γ such that

$$*, \Gamma \models \mathcal{U}.$$

The above definition of instantiation induces a notion of *principal* attribution.

Principal Attribution Theorem 8.21 *Let $\mathcal{U}$ be a system of uniqueness requirements. It is decidable whether a given (conventional) solution of $\mathcal{U}$ is attributable. Moreover, if this is the case, a principal attribution can be computed.*

Proof. [Sketch of the algorithm] Given $*_0 \models |\mathcal{U}|$, lift $*_0$ to a uniqueness type substitution $*$ by choosing fresh attribute variables at each subtype, such that outermost attributes correspond to the attributes of variables in $\mathcal{U}$. Now compute $attr(\mathcal{U}^*)$. If this gives fail, then $*_0$ is not attributable. Otherwise (say $attr(\mathcal{U}^*) = \Gamma$) output $(*, \Gamma)$. □

Analogous to the situation in conventional typing, we formulate a notion of 'exactness' to express that a certain system precisely captures the minimal uniqueness requirements needed for a valid typing.

Definition 8.22 A system of uniqueness requirements $\mathcal{U}$ is called *uniqueness exact* for B, E, S if for all $*, \Gamma$

(1) $*, \Gamma \models \mathcal{U} \;\Rightarrow\; B^* \vdash^{\mathrm{sd}} E : S^* \mid \Gamma$.

(2) $B^* \vdash^{\mathrm{sd}} E : S^* \mid \Gamma \;\Rightarrow\; *', \Gamma' \models \mathcal{U}$
for some $*', \Gamma'$ such that $*', \Gamma' \sim_{B,S} *, \Gamma$.

Proposition 8.23 *Let E be an expression, B a basis and S a uniqueness type. Then there exists a finite system of requirements $\mathcal{U} = \mathcal{U}(B, E, S)$, computable from B, E, S, such that $\mathcal{U}$ is exact for B, E, S.*

Proof. The following inductively defines $\mathcal{U}(B, E, S)$. Union of requirement systems is

to be taken componentwise.

$$
\begin{array}{rcl}
\mathcal{U}(B, x^{\odot}, S) & = & \langle \{B(x) \leq S\}, \emptyset \rangle, \\
\mathcal{U}(B, x^{\otimes}, S) & = & \langle \{B(x) \leq S\}, \{\ulcorner S \urcorner = \times\} \rangle, \\
\mathcal{U}(B, \mathbf{S}\vec{E}, S) & = & \bigcup_i \mathcal{U}(B, E_i, T_i) \cup \langle \{R \preceq S\}, \Delta \rangle \\
 & & \text{if } \vec{T} \rhd R \mid \Delta \text{ is the standard } \mathcal{F},\mathcal{A}\text{-type of } \mathbf{S}, \\
\mathcal{U}(B, \mathsf{let}\ x = E\ \mathsf{in}\ E', S) & = & \mathcal{U}(B \cup x{:}\alpha^a, E', S) \cup \mathcal{U}(B, E, \alpha^a) \\
 & & \alpha, a \text{ fresh}, \\
\mathcal{U}(B, \mathsf{letrec}\ \vec{x} = \vec{E}\ \mathsf{in}\ E', S) & = & \bigcup_i \mathcal{U}(B \cup \vec{x}{:}\vec{\alpha^a}, E_i, {\alpha_i}^{a_i}) \cup \mathcal{U}(B \cup \vec{x}{:}\vec{\alpha^a}, E', S) \\
 & & \vec{\alpha}, \vec{a} \text{ fresh}, \\
\mathcal{U}(B, \mathsf{case}\ E\ \mathsf{of}\ \vec{P}|\vec{E}, S) & = & \mathcal{U}(B, E, \alpha^a) \cup \\
 & & \qquad \bigcup_i (\mathcal{U}(B \cup \vec{x_i}{:}\vec{T_i}, E_i, S) \cup \langle \{\alpha^a \simeq R_i\}, \Delta_i \rangle), \\
 & & \text{if } P_i = \mathbf{C}_i\vec{x_i}; \text{ the standard type of } \mathbf{C}_i \text{ is } \vec{T_i} \rhd R_i \mid \Delta_i, \\
 & & \alpha, a \text{ fresh}.
\end{array}
$$

The verification is similar to the one in the conventional case. As to variables $x^{\otimes}$, note that

$$[S]_\Gamma \leq_\Gamma S' \quad \Leftrightarrow \quad S \leq_\Gamma S',\ \Gamma \vdash \ulcorner S' \urcorner = \times. \ \square$$

As in the conventional case, the typing procedure is started with parameters that do not impose any restriction on solutions, cf. the proof of Theorem 5.20.

Definition 8.24 Let E be an expression, say with $\mathrm{FV}(E) = \{x_1, \ldots, x_n\}$. The *basic system* for E is the system $\mathcal{U}(E) = \mathcal{U}(B_0, E, S_0)$, where $B_0 = \{x_1{:}{\alpha_1}^{a_1}, \ldots, x_n{:}{\alpha_n}^{a_n}\}$ and $S_0 = \alpha^a$.

The above discussion suggests the following strategy for determining uniqueness types. Given E, determine a (conventional) solution $*_0$ of $|\mathcal{U}(E)|$; then use the Principal Attribution Theorem to lift this solution to a uniqueness typing. If this succeeds (say with output $*, \Gamma$), then conclude $B_0^* \vdash_\Gamma E : S_0^*$.

A natural attempt is to take the most general conventional solution for $*_0$. However, because of our treatment of higher-order functions (involving a restriction on the subtype relation w.r.t. variables), it might be the case that lifting this most general solution fails, whereas some specific instance is attributable. Therefore, a reasonable notion of 'most general solution' cannot be formulated for the *combination* $*, \Gamma$. Consequently, there is no 'Principal Uniqueness Type Theorem'. Instead, we stick to the asymmetric approach suggested by the notion of principal attribution with respect to a given (previously determined) substitution. This is reflected in the description of the typing procedure in *Clean*.

Uniqueness Type Inference in Clean

In order to translate the above into a suitable actual typing algorithm we indeed try to lift the most general solution of $|\mathcal{U}(E)|$. If this attempt fails, however, we do not try any specific instances but consider the expression untypable.

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

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

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

In fact, the *Clean* compiler adopts the Hindley-Milner approach towards recursion: in the definition of, say, **F**, all occurrences of **F** should be typed with **F**'s environment type (i.e., without instantiation). Indirect recursion is treated similarly. This can be achieved, for instance, by adjusting the definition of $\mathcal{U}(B, E, S)$ for $E = \mathbf{S}\vec{E}$ in the proof of Proposition 8.23.

Alternative reference count analysis

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

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

$$\begin{array}{rlll} \mathbf{Cond}(b, x, y) = \mathsf{case}\ b\ \mathsf{of} & \mathbf{True} & | & x \\ & \mathbf{False} & | & y \end{array}$$

and compute $\mathbf{Cond}(B, E, E')$ in the standard way, the condition is evaluated first (with possible sharing between B and E, E') and subsequently one of the alternatives is chosen and evaluated (so sharing between E and E' has disappeared). This suggests that we can distinguish between references to the same object inside B, E and E' respectively, allowing a less restrictive uniqueness typing.

Note that the syntax directed system of Definition 8.14 provides the possibility to have different markings of the same variable. In view of the above analysis, the following expression is well-marked:

$$\mathsf{let}\ x = \mathbf{A}\ \mathsf{in}\ \mathbf{Cond}(\mathbf{F}(x^{\otimes}), \mathbf{G}(x^{\odot}), \mathbf{H}(x^{\odot})).$$

In fact, the results of Barendsen and Smetsers (1993) already abstract from the way references are counted: they capture both the standard and the refined approach.

9. Conclusions and Related Work

We have developed a very powerful type system in natural deduction style. Its aim is 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. Both soundness (with respect to a graph rewriting semantics) and decidability of resulting system have been shown.

The system has been implemented as part of the *Clean*-compiler.

The present work has been inspired by Guzmán and Hudak (1990), addressing the mutability problem in a 'single threaded polymorphic lambda calculus' (*poly*-λ_{st}). Instead of using an operational semantics directly based on graph rewriting, they apply lambda-graph reduction due to Wadsworth (1971). Type reconstruction is (roughly) described by defining a type inference algorithm, refraining from a formal correctness proof.

In Turner et al. (1995), a type system is presented that is strongly related to ours. However, the design was guided by different motives: its main purpose is to deal with program transformations (in particular, inlining of unique expressions) and with superfluous closure updates. The main difference with our system is that uniqueness (*use* 1, using their terminology) is not a property of a reference to an expression but rather of the expression itself. The subtyping relation (in our system needed to adjust an offered argument type to the corresponding requested type) is absent, since it would destroy the intended uniqueness property.

In spite of the fundamental efforts of Guzmán and Hudak (1990) and Turner et al. (1995), none of the described approaches has been turned into a real implementation.

Type systems with subtyping have, among others, been studied in Mitchell (1991), presenting a type inference algorithm which determines the minimal set of coercions necessary to type a given term. The algorithm essentially derives the same set of coercions as our procedure introduced at the end of section 8. However, due to our consistency requirement and the subtyping restriction on arrow types (and on type variables), the former may lead to a collection containing unrealizable coercions. It needs to be investigated whether it is decidable if a given expression is uniqueness typable, even though there need not be a 'minimal' uniqueness type (see the discussion in Section 8).

Hankin and le Métayer (1994) present a general method for deriving type inference algorithms from (non-standard) type systems. The main application of this method is strictness analysis. It would be an interesting experiment to formulate uniqueness typing in this framework, and to compare the resulting type inference algorithm with the one described in the present paper.

In any case, we are planning to investigate whether our way of (non-standard) typing and type inference applies to other areas of static analysis, such as strictness analysis.

As has been mentioned before, our system is closely connected to substructural logics. A combined linear/full intuitionistic logic can be found in Benton (1994). The system described in this paper could be the first step towards a 'propositions as types/proofs as graphs' notion.

References

Ariola, Z.M. and J.W. Klop (1995). Equational term graph rewriting, *Technical Report CS-R9552*, Centrum voor Wiskunde en Informatica (CWI), Computer Science/Department of Software Technology.

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. (1992). Lambda calculi with types, *in:* S. Abramsky, D.M. Gabbay and T.S.E. Maibaum (eds.), *Handbook of Logic in Computer Science*, Vol. II, Oxford University Press.

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

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

Barendsen, E. and J.E.W. Smetsers (1993). Conventional and uniqueness typing in graph rewrite systems (extended abstract), *in:* R.K. Shyamasundar (ed.), *Proceedings of the 13th Conference on Foundations of Software Technology and Theoretical Computer Science*, Bombay, India, Lecture Notes in Computer Science 761, Springer-Verlag, Berlin, pp. 41–51. Full paper: see Technical Report CSI-R9328, University of Nijmegen, and Barendsen (1995).

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

Barendsen, E. and J.E.W. Smetsers (1995a). A derivation system for uniqueness typing, *in:* A. Corradini and U. Montanari (eds.), *SEGRAGRA'95: Joint Compugraph/Semagraph Workshop on Graph Rewriting and Computation*, Volterra (Pisa), Italy, Electronic Notes in Theoretical Computer Science, Elsevier Science, pp. 151–158.

Barendsen, E. and J.E.W. Smetsers (1995b). Uniqueness type inference, *in:* M. Hermenegildo and S.D. Swierstra (eds.), *Programming Languages: Implementations, Logics and Programs (PLILP'95)*, Utrecht, The Netherlands, Lecture Notes in Computer Science 982, Springer-Verlag, Berlin, pp. 189–206.

Barendsen, E. and J.E.W. Smetsers (1995c). Uniqueness typing in natural deduction style (extended abstract), *1995 Glasgow Workshop on Functional Programming*, pp. XVI 1–XVI 10. Accepted for the formal proceedings (to appear).

Benton, P.N. (1994). A mixed linear and non-linear logic: Proofs, terms and models, *in:* L. Pacholski and J. Tiuryn (eds.), *Computer Science Logic, 8th Workshop*, Kazimierz, Poland, Lecture Notes in Computer Science 933, Springer-Verlag, Berlin, pp. 121–135.

Blass, A. (1992). A game semantics for linear logic, *Annals of Pure and Applied Logic* **56**, pp. 183–220.

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.

Hankin, C. and D. le Métayer (1994). Deriving algorithms from type inference systems: Applicaton to strictness analysis, *POPL'94: 21st ACM SIGPLAN-SIGACT Symposium of Principles of Programming Languages*, Portland, Oregon, ACM Press, pp. 202–213.

Mitchell, J.C (1991). Type inference with simple subtypes, *Journal of Functional Programming* **1**, pp. 245–285.

Plasmeijer, M.J. and M.C.J.D. van Eekelen (1995). Concurrent Clean. Available via `www.cs.kun.nl/~clean/`.

Reynolds, J.C. (1995). Passivity and linear types. Talk given at the conference on Types for Proofs and Programs, Turin, Italy, June 1995.

Robinson, J.A. (1965). A machine-oriented logic based on the resolution principle, *Journal of the Association for Computing Machinery* **12**, pp. 23–41.

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

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

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

Wand, M. (1987). A simple algorithm and proof for type inference, *Fundamenta Informaticae* **X**, pp. 115–122.