Type Systems for Graph Rewriting

Erik Barendsen Sjaak Smetsers
University of Nijmegen*

Abstract

We present two type systems for 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.

Due to the present formalization, the system can easily be compared with other proposals based on linear and affine logic.

1. Introduction

In recent years, various proposals have been brought up as solutions to the —at first sight paradoxical— desire to allow destructive operations in a functional context.

Indeed, by admitting these operations (such as file manipulations) one loses referential transparency. The essence of common solutions (e.g., Wadler (1990), Guzmán and Hudak (1990)) is the restriction of destructive operations to 'local' input objects (occurring only once).

The uniqueness type system for graph rewrite systems (presented in Barendsen and Smetsers (1993a) and (1993b)) offers the possibility to indicate such locality requirements of functions in the types of the corresponding arguments. These special so-called *uniqueness types* are annotated versions of traditional Curry-like types. E.g. the operation **WriteChar** which writes a character to a file is typed with **WriteChar** : $(\mathsf{Char}^{\times}, \mathsf{File}^{\bullet}) \rightsquigarrow \mathsf{File}^{\bullet}$. Here, $\bullet$, $\times$ stand for 'unique' and 'non-unique' respectively.

Uniqueness typing can be regarded as a combination of linear typing (dealing with unique objects) and traditional typing (for non-unique objects, without restrictions on their reference counts), 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).

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

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 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 denoted by $[\![E]\!]$. In contrast with Ariola and Klop (1995), we refrain from defining a reduction relation on the expressions themselves.

The paper is organized as follows. After a very short and informal introduction to (Term) Graph Rewriting (Section 2), we define a formal language for denoting graph expressions and rewrite rules (Section 3). In Section 4 a Curry-like (conventional) type system is introduced. 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). For both system we prove preservation of typing during reduction, and the existence of principal types (Sections 5 resp. 6 for conventional typing, and Sections 9 resp. 10 for uniqueness typing). In Section 11 we describe how uniqueness type inference proceeds in the functional programming language *Clean* (see also Plasmeijer and van Eekelen (1995) and conclude with a discussion of related work (Section 12).

The original uniqueness type system is rather complex, particularly due to the refined reference analysis (for details, see Barendsen and Smetsers (1993a)). 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.

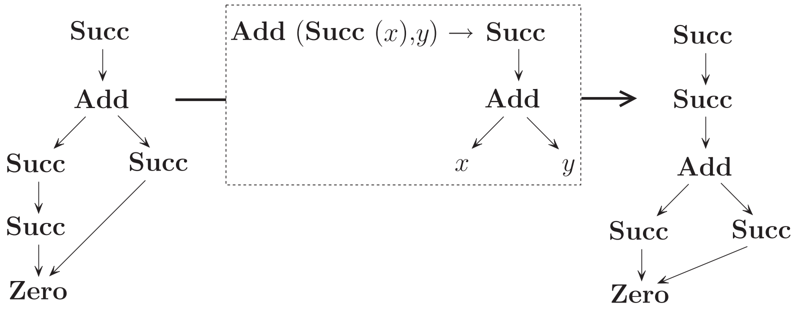
2. Graph Rewriting

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

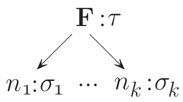
Term graph rewriting can be seen as an extension of term rewriting with sharing. In term rewrite rules, multiple occurrences of variables in lead to duplication of actual instances. In GRS's, such duplications are avoided by copying references to the objects instead of copying the objects themselves.

The objects in GRS'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 (the *arguments*) of any node labelled with $\mathbf{S}$. Rewrite rules 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 say that this rule is a *function/constructor rule* if $\vec{p}$ contains only data constructors. In the present paper we restrict ourselves to so-called *left-linear function/constructor systems.*

The following picture shows the effect of rewriting a graph according to the displayed rule.



As in van Bakel et al. (1992), a notion of type assignment is defined by supplying nodes of the graphs with types in a consistent way. Here, consistent means that for any application of the form



the combination of the types $\vec{\sigma}$ (assigned to the arguments $n_1, \ldots, n_k$) and the type τ (assigned to the $\mathbf{F}$-node itself) should be an instance of $\mathbf{F}$'s rule type.

3. Syntax

We present a syntax of a graph-like formal language which incorporates some essential aspects of graph rewriting: sharing, cycles and pattern matching. The *objects* are terms generated by the following syntax.

$$\begin{array}{lcl} E & ::= & x \mid \mathbf{S}(E_1, \ldots, E_k) \mid \mathsf{let}\ x = E\ \mathsf{in}\ E' \mid \mathsf{letrec}\ \vec{x} = \vec{E}\ \mathsf{in}\ E' \mid \mathsf{case}_{\mathsf{T}} E\ \mathsf{of}\ \vec{P} | \vec{E}, \\ P & ::= & \mathbf{C}(x_1, \ldots, x_k). \end{array}$$

Here $x, \vec{x}$ range over (sequences of) term variables, and $\mathbf{S}$ over some set of *symbols* of fixed arity (we will sometimes suggestively use $\mathbf{F}$ for functions and $\mathbf{C}$ for data constructors). Data constructors are assumed to introduced by a so-called *algebraic type system* $\mathcal{A}$. $\mathcal{A}$ contains specifications like

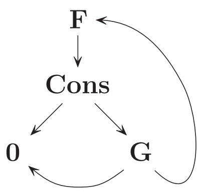
$$\mathsf{List}(\alpha) = \mathbf{Cons}(\alpha, \mathsf{List}(\alpha)) \mid \mathbf{Nil}$$

declaring the data constructors **Cons** and **Nil**, and moreover the type constructor List.

Ordinary sharing can be expressed using a let construct, whereas letrec introduces cyclic dependencies. E.g., 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 graph



To be able to decompose data objects, a special case_T construct is associated with each data type declaration in $\mathcal{A}$. An application of case_T contains alternative specifications for each data constructor of type T. The patterns of such an application (indicated by P in the above syntax) are supposed to be *linear*: no variable is occurring more then once in the same pattern.

Rewrite rules are denoted using *function definitions* of the form

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

For example,

$$\mathbf{Append}(\ell_1, \ell_2) \to \mathsf{case}_{\mathsf{List}}\, \ell_1 \;\mathsf{of}\; \begin{array}{l|l} \mathbf{Cons}(h,t) & \mathbf{Cons}(h, \mathbf{Append}(t, \ell_2)) \\ \mathbf{Nil} & \ell_2. \end{array}$$

4. Conventional Typing

For the sequel, fix some algebraic system $\mathcal{A}$. Types (in $\mathcal{A}$) are built up from type variables and type constructors.

$$\sigma \;::=\; \alpha \mid \mathsf{T}\vec{\sigma}.$$

The function symbols are supplied with a type by a (function) *type environment* $\mathcal{F}$. Such an environment contains declarations

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

where k is the arity of $\mathbf{F}$. In the sequel, fix some function type environment $\mathcal{F}$.

Due to the separation of specifications (function definitions, algebraic types) from applications one needs an *instantiation mechanism* to deal with different occurrences of symbols.

The algebraic type system $\mathcal{A}$ gives the types of the algebraic constructors. Let

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

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

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

For example, for lists one has

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

For function types we have

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

if $\mathcal{F}$ contains $\mathbf{F} : \vec{\sigma} \rightsquigarrow \tau$. Instantiation of symbol types is defined as follows ($[\alpha := \rho]$ denotes substitution).

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

Our system deals with *typing statements* of the form

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

where B is some finite set of variable declarations of the form $x{:}\tau$ called a *basis*. Such a statement is valid if it can be produced using the following derivation rules.

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

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

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

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

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

This concludes the treatment of expressions. If we consider rewrite rules, the environment $\mathcal{F}$ should be consistent with these rules. We say that the rule

$$\mathbf{F}\vec{x} \to E,$$

say with $\mathcal{F}$ containing $\mathbf{F}{:}\vec{\sigma} \leadsto \tau$, is *type correct* if

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

5. Semantics

We now give a formal account of the (intuitively clear) interpretation of expressions and function definitions as graphs and rewrite rules respectively. The terminology will be used in the proof of the fact that the derivation system is sound with respect to graph rewriting.

We will 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 example graph in Section 3 can be denoted by

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

In general, 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 free variables of g (notation $\mathrm{FV}(g)$) consists of those in $\mathrm{V}(g)$ that do not appear as a left-hand side of an equation in G; the other (so-called bound) variables are indicated by $\mathrm{BV}(g)$. A graph is *closed* if $\mathrm{FV}(g) = \emptyset$.

Now the notion of typing for graphs (see Section 4) can now be be reformulated: a *typing* for $g = \langle r \mid G\rangle$ is a function $\mathcal{T} : \mathrm{V}(g) \to \mathbb{T}$ such that for any equation $x = \mathbf{S}(\vec{y})$ in G

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

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 $\mathcal{T}(g) = \sigma$.

5.1. Definition. 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) $\vec{x}$.

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

We have omitted the $\mathsf{case}_{\mathsf{T}}$ construct: it will be included when dealing with rewrite rules.

5.2. Definition. 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

$$\mathcal{T} \models B \;\;\&\;\; \mathcal{T} \models E : \sigma.$$

5.3. Proposition (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 $B \vdash E : \sigma$. $\square$

5.4. Proposition (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. $\square$

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

5.6. COROLLARY. $B \vdash E : \sigma,\ E \sim E' \ \Rightarrow\ B \vdash E' : \sigma$

Typing of graph rewrite rules (cf. Barendsen and Smetsers (1993a)) can be formulated as follows.

5.7. DEFINITION. (i) A type assignment $\mathcal{T}$ to variables can be extended to patterns in a straightforward way:

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

(ii) The rewrite rule $\mathbf{F}\vec{p} \to g$ is *type correct* in $\mathcal{F}$ (containing $\mathbf{F}{:}\vec{\sigma} \rightsquigarrow \tau$) if for some $\mathcal{T}$ one has

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

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

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

SUBJECT REDUCTION THEOREM.

$$\left.\begin{array}{r} g : \sigma, \\ \mathcal{R}\ \textit{is type correct}, \\ g \twoheadrightarrow_{\mathcal{R}} g' \end{array}\right\} \quad \Rightarrow \quad g' : \sigma.$$

We now turn to the interpretation of function definitions as graph rewrite rules. To simplify the description, case expressions are only allowed in the right-hand sides of rewrite rules. Moreover, we restrict applications of case to variables only, and we do not allow them inside other expressions. Clearly, these restrictions do not limit definability of functions: it is always possible to translate a function with multiple, possibly nested patterns into the requested form (by adding new auxiliary rules for each extra pattern). By these restrictions, the translation of function definitions to rewrite rules is quite simple.

5.8. DEFINITION. (i) The *graph interpretation* of 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:

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

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

$$[\![\mathbf{F}(\vec{x}, y) = \mathsf{case}_{\mathsf{T}} y \ \mathsf{of}\ \vec{P} \mid \vec{E}]\!] \quad = \quad \left\{\begin{array}{lcl} \mathbf{F}\vec{x}P_1 & \to & [\![E_1]\!], \\ & \vdots & \\ \mathbf{F}\vec{x}P_n & \to & [\![E_n]\!]. \end{array}\right.$$

(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.

5.9. LEMMA. $\mathbb{F}$ *is type correct* $\Rightarrow$ $[\![\mathbb{F}]\!]$ *is type correct.*

Combining the above results we can formulate the soundness of the expression type system with respect to graph rewriting semantics.

5.10. THEOREM.

$$\left.\begin{array}{r} B \vdash E : \sigma, \\ \mathbb{F} \textit{ is type correct}, \\ [\![E]\!] \xrightarrow[[\![\mathbb{F}]\!]]{} [\![E']\!] \end{array}\right\} \quad \Rightarrow \quad B \vdash E' : \sigma.$$

6. Principal Types

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.

6.1. DEFINITION. Let $\mathcal{E} = \{\sigma_1 = \tau_1, \ldots, \sigma_n = \tau_n\}$ be a finite set of equations between types. A *solution* for $\mathcal{E}$ is a substitution $*$ such that

$$\sigma_1^* = \tau_1^* \ \& \ \cdots \ \& \ \sigma_n^* = \tau_n^*.$$

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

6.2. UNIFICATION THEOREM. *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). $\square$

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.

6.3. DEFINITION. Let E be an expression, B a basis, and $\sigma \in \mathbb{T}$. A set of type equations $\mathcal{E}$ is *exact* for B, E, σ if for each substitution $*$ one has

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

(2) $B^* \vdash E : \sigma^* \quad \Rightarrow \quad *' \models E$ for some $*'$ such that $*' \upharpoonright (B, \sigma) = * \upharpoonright (B, \sigma)$.

6.4. PROPOSITION. *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:

$$\begin{array}{rcl}
\mathcal{E}(B, x, \sigma) & = & \{B(x) = \sigma\} \quad \text{(regard } B \text{ as a partial function)}, \\
\mathcal{E}(B, \mathbf{S}\vec{E}, \sigma) & = & \bigcup_i \mathcal{E}(B, E_i, \tau_i) \cup \{\rho = \sigma\} \\
 & & \text{if } \mathcal{F}, \mathcal{A} \text{ contains } \mathbf{S} : \vec{\tau} \rightsquigarrow \rho, \\
\mathcal{E}(B, \mathsf{let}\ x = E\ \mathsf{in}\ E', \sigma) & = & \mathcal{E}(B, x : \alpha, E', \sigma) \cup \mathcal{E}(B, E, \alpha) \\
 & & \alpha \text{ fresh}, \\
\mathcal{E}(B, \mathsf{letrec}\ \vec{x} = \vec{E}\ \mathsf{in}\ E', \sigma) & = & \mathcal{E}(B, \vec{x} : \vec{\alpha}, E', \sigma) \cup \bigcup_i \mathcal{E}(B, \vec{x} : \vec{\alpha}, E_i, \alpha_i) \\
 & & \vec{\alpha} \text{ fresh}, \\
\mathcal{E}(B, \mathsf{case}_\mathsf{T} E\ \mathsf{of}\ \vec{P}|\vec{E}, \sigma) & = & \mathcal{E}(B, E, T\vec{\alpha}) \cup \bigcup_i \mathcal{E}(B, \vec{x_i} : \vec{\sigma_i}, E, \sigma) \\
 & & \text{if } \mathcal{A} \text{ contains } \mathsf{T}\vec{\alpha} = \mathbf{C}_1\vec{\sigma_1} \mid \cdots.
\end{array}$$

One shows (1) by induction on E. We will only describe the case $E = \mathbf{S}\vec{E}$; all other cases are treated similarly. Suppose $E = \mathbf{S}\vec{E}$. Then $\mathcal{E}(B, E, \sigma) = \bigcup_i \mathcal{E}(B, E_i, \tau_i) \cup \{\rho = \sigma\}$, where $\mathcal{F}, \mathcal{A}(S) = \vec{\sigma} \rightsquigarrow \rho$. Since $* \models \mathcal{E}(B, E, \sigma)$ one has $* \models \mathcal{E}(B, E_i, \tau_i)$ and $* \models \sigma = \rho$. Then $B^* \vdash E_i : \tau_i^*$ by induction hypothesis. Moreover, $\sigma^* = \rho^*$, so $\mathcal{F}, \mathcal{A} \vdash \mathbf{S} : \vec{\tau^*} \rightsquigarrow \sigma^*$. Applying the application rule yields $B^* \vdash E : \sigma^*$.

As to (2), use induction on the derivation of $B^* \vdash E : \sigma^*$. Again we will describe only one example case: the sharing rule. So assume $B^* \vdash \mathsf{let}\ x = E\ \mathsf{in}\ E' : \sigma^*$ since $B^* \vdash E : \tau$ and $B^*, x : \tau \vdash E' : \sigma^*$. From the definition of $\mathcal{E}$ it follows that $\mathcal{E}(B, \mathsf{let}\ x = E\ \mathsf{in}\ E', \sigma^*) = \mathcal{E}(B, E, \alpha) \cup \mathcal{E}(B, x : \alpha, E, \sigma^*)$. Set $*_1 = [\alpha := \tau] \circ *$. Note that since α is fresh $*_1 \upharpoonright (B, \sigma) = * \upharpoonright (B, \sigma)$. Moreover, $B^{*_1} \vdash E : \alpha^{*_1}$ and $(B, x : \alpha)^{*_1} \vdash E' : \sigma^{*_1}$. Applying the induction hypothesis twice gives us $*_2 \models \mathcal{E}(B, E, \alpha)$ for some $*_2$ such that $*_2 \upharpoonright (B, \alpha) = *_1 \upharpoonright (B, \alpha)$ and $*_3 \models \mathcal{E}(B, x : \alpha, E', \sigma)$ for some $*_3$ such that $*_3 \upharpoonright (B, \alpha, \sigma) = *_1 \upharpoonright (B, \alpha, \sigma)$. Now observe that apart from the type variables in B, α the substitution $*_2$ only affects 'local' (i.e. fresh) variables in $\mathcal{E}(B, E, \alpha)$. The same holds for $*_3$ w.r.t. B, E, α and σ. Hence

$$*_2 \cup *_3 \models \mathcal{E}(B, E, \alpha) \cup \mathcal{E}(B, x : \alpha, E', \sigma)$$

and moreover

$$*_2 \cup *_3 \upharpoonright (B, \sigma) = *_1 \upharpoonright (B, \sigma) = * \upharpoonright (B, \sigma). \ \square$$

6.5. DEFINITION. Let E be an expression. The pair (B, σ) is a *principal pair* for E if

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

PRINCIPAL TYPE THEOREM. *There exists a recursive function pt such that*

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

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

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

Suppose E is typable. Suppose $B \vdash E : \sigma$. Say $B \supset \{x_1{:}\sigma_1, \ldots, x_k{:}\sigma_k\}$. Set $*_1 = [\alpha := \sigma, \alpha_1 := \sigma_1, \ldots, \alpha_k := \sigma_k]$. Note that $B_0^{*_1} \subseteq B$ and $\sigma_0^{*_1} = \sigma$, so $B_0^{*_1} \vdash E : \sigma_0^*$. By proposition 6.4 (2) there exists $*'_1$ such that $*'_1 \models \mathcal{E}(B_0, E, \sigma_0))$ and $*'_1 \restriction (B_0, \sigma_0) = *_1 \restriction (B_0, \sigma_0)$. Consequently, $\mathrm{Unify}(\mathcal{E}(B_0, E, \sigma_0)) = *$ for some proper $*$ and

(1) $B_0^* \vdash E : \sigma_0^*$ by Proposition 6.4;

(2) $*'_1 = *_2 \circ *$ for some $*_2$ by Theorem 6.2.

Combining (1) and (2) gives $B_0^{*_1} = B_0^{*'_1} = (B_0^*)^{*_2}$, and $\sigma_0^{*_1} = (\sigma_0^*)^{*_2}$. If E is not typable, then there exists no $*$ such that $* \models \mathcal{E}(B, E, \sigma)$, and hence

$$\mathrm{Unify}(\mathcal{E}(B, E, \sigma)) = \mathsf{fail} = pt(E). \ \square$$

7. Simple Uniqueness Typing

Uniqueness typing offers the possibility to indicate locality requirements of functions in the correspoding argument types. In particular, it can be specified that a given argument should be unique (by the annotation $\bullet$). In any application, the concrete function argument should have reference count 1. So the function has indeed 'private' access to its argument, and hence the argument can be updated in-place. This is used for inherently destructive operations, like **WriteChar**, but also to improve storage management. Take, for instance, the following list reversing function which can be implemented as a 'destructive' function if the given uniqueness type is used.

$\mathbf{Rev} : \mathsf{List}^\bullet(\alpha^\times) \rightsquigarrow \mathsf{List}^\bullet(\alpha^\times)$

$\mathbf{Rev}\ (l) \quad \rightarrow \quad \mathbf{H}(l, \mathbf{Nil})$

$\mathbf{H} : (\mathsf{List}^\bullet(\alpha^\times), \mathsf{List}^\bullet(\alpha^\times)) \rightsquigarrow \mathsf{List}^\bullet(\alpha^\times)$

$$\begin{array}{lcllll} \mathbf{H}(\ell_1, \ell_2) & \rightarrow & \mathsf{case}_{\mathsf{List}}\ \ell_1\ \mathsf{of} & \mathbf{Cons}(h, t) & \mid & \mathbf{H}(t, \mathbf{Cons}(h, \ell_2)) \\ & & & \mathbf{Nil} & \mid & \ell_2. \end{array}$$

Note that not only the space of the obsolete **Cons** can be re-used, but also parts of its contents. In fact it already suffices to change the reference to t to point to ℓ_2. Such re-usage of space is often called *compile-time garbage collection.*

Uniqueness typing combines conventional typing and linear typing, through a reference count administration. An environment type $\mathbf{F} : \sigma^\bullet \rightsquigarrow \cdots$ means that $\mathbf{F}$'s argument should be unique for $\mathbf{F}$, i.e., should have reference count 1. In the same way, uniqueness of results is specified: if $\mathbf{G} : \cdots \rightsquigarrow \sigma^\bullet$, then a well-typed expression $\mathbf{F}(\mathbf{G}(E))$ remains type-correct, even if $\mathbf{G}(E)$ is subject to computation. Sometimes, uniqueness is not required. If $F : \sigma^\times \rightsquigarrow \cdots$ then still $\mathbf{F}(\mathbf{G}(E))$ is type correct. This is expressed in a subtype relation, such that roughly $\sigma^\bullet \le \sigma^\times$. Offering a non-unique argument if a function requires a unique one fails: $\sigma^\times \not\le \sigma^\bullet$. The subtype relation is defined in terms of the ordering $\bullet \le \times$ on attributes. In an application an argument can also be non-unique if has reference count greater than 1 (even though the type of the argument expression itself is unique). This is covered by a correction mechanism connecting non-unique ('conventional') and unique ('linear') types: a unique result may be used more than once, as long as only non-unique supertypes are required.

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:

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

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 (1993a).

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

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

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).

Algebraic Uniqueness Types

We will first describe the system without the type constructor $\rightarrow$. Below, $S, T, \ldots$ range over uniqueness types. The outermost attribute of S is denoted by $\ulcorner S \urcorner$.

The subtype relation 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: if the object in question has type S, then only non-unique versions of S may be used. Given S, we construct the smallest non-unique supertype of S:

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

Cyclic objects (with their inherent sharing) are treated by correcting both internal and external references to their roots.

The environment $\mathcal{F}$ contains declarations $\mathbf{F} : \vec{S} \rightsquigarrow T$. Symbol types are extracted from $\mathcal{F}, \mathcal{A}$ in the following way. Say the algebraic environment $\mathcal{A}$ contains

$$T\vec{\alpha} = C_1\vec{\sigma_1} | \cdots$$

Then for any well-attributed variant $\vec{S}, T$ of $\vec{\sigma_i}, \mathsf{T}\vec{\alpha}$ respectively

$$\mathcal{A} \vdash C_i : \vec{S} \rightsquigarrow T.$$

Moreover

$$\mathcal{F}, \mathbf{F}{:}\vec{S} \rightsquigarrow T \vdash \mathbf{F} : \vec{S} \rightsquigarrow T.$$

Symbol types are instantiated via the rule

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

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

$$B \vdash E : S.$$

In our language, sharing appears as multiple occurrences of the same variable. Like in linear logic, we have to be precise when dealing with bases used for typing subterms: the denotation B_1, B_2 stands for a disjoint union of bases.

The rules for type assignment are the following.

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

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

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

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

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

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

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.

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

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

Like in the conventional case, $\mathcal{F}$ should be consistent with the rewrite rules. A rule

$$\mathbf{F}\vec{x} \to E,$$

say with $\mathcal{F}$ containing $\mathbf{F} : \vec{S} \leadsto T$, is *type correct* if

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

Higher-Order Uniqueness Types

We will now describe the incorporation of *higher-order functions*. Higher-order functions give rise to partial (often called *Curried*) symbol applications. In functional programming languages, these applications are usually written as $\mathbf{F}E_1 \cdots E_k$ (with $k < \text{arity}(\mathbf{F})$), denoting the function $\lambda\vec{x}.\mathbf{F}(E_1, \ldots, E_k, x_{k+1}, \ldots, x_{\text{arity}(\mathbf{F})})$.

If these partial applications contain unique subexpressions one has to be careful. Consider, for example, a function $\mathbf{F}$ with type $\mathbf{F} : (\sigma^\bullet, \tau^\times) \leadsto \sigma^\bullet$ in the application $\mathbf{F}E$. Clearly, the result type of this application is of the form $\tau^\times \stackrel{u}{\to} \sigma^\bullet$. If one allows that this application is used more than once, one cannot guarantee 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. There are several ways to prevent such an expression from being copied. For instance, one might introduce a new uniqueness attribute, say $\vartriangle$, for any unique object that does not coerce to a non-unique variant. This has been described in Barendsen and Smetsers (1993a). Another solution is the region-administration introduced by Reynolds (1995).

Instead of introducing a new attribute, the present paper considers the $\to$ 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 $\to$ 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. E.g.

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

Notice the so called contravariance and covariance of the $\to$ constructor: one says that the first argument occurs on a *negative* position. This is generalized for ordinary type constructors making the subtyping relation dependent on a 'negative/positive' classification of constructor arguments. For a given algebraic type system $\mathcal{A}$ one can deduce such a sign classification by standard (fixedpoint) techniques. For example, suppose T is defined as

$$\mathsf{T}(\alpha, \beta, \gamma) = \mathbf{C}(\mathsf{List}(\beta \rightarrow \alpha)) \mid \mathbf{D}(\gamma \rightarrow \mathsf{List}(\gamma)),$$

then

$$sign(\mathsf{T}) = (\oplus, \ominus, \top).$$

The sign $\top$ indicates that the corresponding argument occurs both on a positive and negative position.

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

Partial applications can be incorporated in our formal system as follows. An application $\mathbf{F} E_1 \cdots E_k$ is written as $\mathbf{F}_k(E_1, \ldots, E_k)$ (so $\mathbf{F}_{\text{arity}(\mathbf{F})} = \mathbf{F}$). Moreover we add the application operator Ap to our syntax with effect (roughly)

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

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

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

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.

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

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

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

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

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

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

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 $u \leq v, v \leq u$. We say that Γ is *consistent* if $\Gamma \nvdash \times \leq \bullet$.

The relation $\leq$ is extended to types. One has, for example,

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

The context rules become

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

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

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

The correction operation is also relativized:

$$\begin{array}{rcll} [\alpha^u]_\Gamma & = & \alpha^u & \text{if } \Gamma \vdash u = \times, \\ [\mathsf{T}^u \vec{S}]_\Gamma & = & \mathsf{T}^\times \vec{S}, & \\ [S \stackrel{u}{\to} T]_\Gamma & = & S \stackrel{u}{\to} T & \text{if } \Gamma \vdash u = \times. \end{array}$$

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 becomes

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

and the subsumption rule

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

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.

8.1. Definition. (i) An environment Γ is *S-equivalent* to Γ' (notation $\Gamma \sim_S \Gamma'$) if for all $u, v \in S \cup \{\bullet, \times\}$

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

(ii) By $B \vdash E : S \mid \Gamma$ we denote that $B \vdash_{\Gamma'} E : S$ for some Γ' with $\Gamma' \sim_{(B,S)} \Gamma$.

Type correctness of rewrite rules can now be expressed as follows. The rule

$$\mathbf{F}\vec{x} \to E,$$

say with $\mathcal{F}$ containing $\mathbf{F} : \vec{S} \rightsquigarrow T \mid \Gamma$, is *type-correct* if

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

9. Soundness of Uniqueness Typing

In this section we will proof that uniqueness typing is sound with respect to graph rewriting.

We start with defining a notion of uniqueness typing for graphs.

A *uniqueness typing* for $g = \langle r \mid G \rangle$ is a pair $\langle \mathcal{U}, \Gamma \rangle$ where $\mathcal{U}$ assigns a uniqueness type to each variable in $\mathrm{V}(g)$ such that for any equation $x = \mathbf{S}(\vec{y})$ in G there exist types $\vec{S}$ with the following properties.

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

and for all $i \leq k$

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

Here the *reference count* $\mathrm{rc}_g(n) = \otimes$ if n appears more than once in the right-hand sides of equations in G, and $\mathrm{rc}_g(n) = \odot$ otherwise. We use $\mathcal{U}(g) = S$ to denote that $\mathcal{U}(r) = S$.

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

Notation. (i) $[S]_\Gamma^\otimes = [S]_\Gamma, [S]_\Gamma^\odot = S$.

(ii) By $\mathcal{U} \models_\Gamma B$ we denote that $\Gamma \vdash \mathcal{U}(x) = S$ for each $(x : S) \in B$.

9.1. Lemma.

$$\Gamma \vdash (S \leq T, \ulcorner T \urcorner = \times) \;\Rightarrow\; \Gamma \vdash [S]_\Gamma \leq T.$$

9.2. Lemma (Contraction Lemma).

$$[\![E[x, y := z]]\!] = [\![E]\!][x, y := z].$$

9.3. PROPOSITION. *Let $B \vdash_\Gamma E : S$. Then there exists a type assignment $\mathcal{U}$ such that $\langle \mathcal{U}, \Gamma \rangle$ is a uniqueness typing for $[\![E]\!]$ with the following properties.*

$$\mathcal{U} \models_\Gamma B$$
$$\Gamma \vdash \mathcal{U}([\![E]\!]) \leq S$$

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

- $\vec{B} \vdash_\Gamma \mathbf{S}\vec{E} : S$ since $\mathcal{F}, \mathcal{A} \vdash \mathbf{S} : \vec{S} \rightsquigarrow T \mid \Gamma$ and $B_i \vdash_\Gamma E_i : S_i$. By I.H. we have $\mathcal{U}_i$ for $[\![E_i]\!]$ with $\mathcal{U}_i \models_\Gamma B_i$ and $\Gamma \vdash \mathcal{U}_i([\![E_i]\!]) \leq S_i$. Say $\langle r_i \mid G_i \rangle = [\![E_i]\!]$. Write $rc_i = \mathrm{rc}_{[\![\mathbf{S}\vec{E}]\!]}(r_i)$. Observe that $rc_i = \otimes$ iff $[\![E_i]\!]$ is cyclic, for, the B_i are disjoint. By Lemma 9.1

$$[\mathcal{U}_i([\![E_i]\!])]_\Gamma^{rc_i} \leq S_i.$$

Set $\mathcal{U} = \bigcup_i \mathcal{U}_i \cup \{s \mapsto T\}$. Then $\langle \mathcal{U}, \Gamma \rangle$ is a uniqueness typing for $[\![S\vec{E}]\!]$ and moreover $\mathcal{U} \models_\Gamma \vec{B}$.

- $B, x{:}S \vdash_\Gamma E[y, z := x] : T$ since $B, y{:}[S]_\Gamma, z{:}[S]_\Gamma \vdash_\Gamma E : T$. By I.H. we have $\mathcal{U} \models_\Gamma B, y{:}[S]_\Gamma, z{:}[S]_\Gamma$ and $\Gamma \vdash \mathcal{U}([\![E]\!]) \leq T$. Set $\mathcal{U}' = \mathcal{U} \cup [x \mapsto S]$. Note that $\mathcal{U}' \models_\Gamma B, x{:}S$. Then by the Contraction Lemma and Lemma 9.1 $\langle \mathcal{U}', \Gamma \rangle$ is a uniqueness typing for $[\![E[y, z := x]]\!]$, and $\Gamma \vdash \mathcal{U}'[\![E[y := x, z := x]]\!] \leq T$. $\square$

For a uniform treatment of all nodes in a graph g (in particular, cyclic nodes) it is sometimes convenient to supply g with a new root. This new root has in-degree 0, and contains a fixed data symbol **Root** of arity 1 referring to the original root of g. The resulting graph is denoted as g^+. Since **Root** is a data symbol, 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.

The type of the **Root** symbol is given by:

$$\mathcal{F}, \mathcal{A} \vdash \mathbf{Root} : \alpha^a \rightsquigarrow \alpha^a.$$

9.4. DEFINITION. Let E be an expression.

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

$$\mathcal{U}([\![E]\!]^+) = S.$$

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

$$\mathcal{U} \models_\Gamma B \;\;\&\;\; \mathcal{U} \models_\Gamma E : S.$$

Using $[\![E]\!]^+$ instead of $[\![E]\!]$ in the latter definition offers the possibility to to perform an extra 'top coercion' of $[\![E]\!]$. This will be used in the following result.

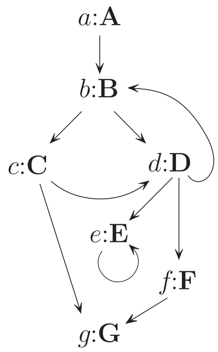
9.5. PROPOSITION (Soundness of expression typing). *For any B, Γ, E, S*

$$B \vdash_\Gamma E : S \;\Rightarrow\; B \models_\Gamma E : S.$$

PROOF. Suppose $B \vdash_\Gamma E : S$. By Proposition 9.3 there exists a type assignment $\mathcal{U}$ such that $\mathcal{U} \models_\Gamma B$ and $\Gamma \vdash \mathcal{U}([\![E]\!]) \leq S$. Say $[\![E]\!]^+ = \langle r^+ \mid G\rangle$. Set $\mathcal{U}' = \mathcal{U} \cup [r^+ \mapsto S]$. Clearly $\mathcal{U}'$ is a typing for $[\![E]\!]^+$. Furthermore, $\mathcal{U}' \models_\Gamma B$ and $\mathcal{U}' \models_\Gamma E : S$.

Showing that the inductive uniqueness type system is powerful enough to capture uniqueness typing of graphs is more subtle than for the conventional case. This is because our typing rules are rather *intentional*: 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. For this reason, the completeness result for uniqueness typing is somewhat more involved.

The idea is to transform a given graph $g = \langle r \mid G\rangle$ into an expression E_g that contains a 'minimal' set 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

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

9.6. DEFINITION. Let $g = \langle r \mid G\rangle$ 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.

$$\begin{array}{rcll} (\mathbf{S}(n_1,\ldots,n_k))^* & = & \mathbf{S}(\overline{n_1},\ldots,\overline{n_k}); & \\ \overline{n} & = & n & \text{if } \mathrm{rc}_g(n) = \otimes \text{ or } n \in \mathrm{FV}(g) \\ & = & {E_n}^* & \text{if } \mathrm{rc}_g(n) = \odot \text{ and } n \in \mathrm{BV}(g) \end{array}$$

(ii) Finally, the *standard* expression denoting g (notation E_g) is given by

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

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.

9.7. DEFINITION. Let $n \in V(g)$. The 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.

9.8. LEMMA. $[\![E_g]\!] = g$

PROOF. Obvious. $\square$

Below we will use the degree as a technical tool to proof that E_g can be typed with the same type as g. In the sequel, fix a graph g and an uniqueness typing $\langle \mathcal{U}, \Gamma \rangle$ for g.

The aim is to show the following result.

9.9. THEOREM (Completeness for expression uniqueness typing).

$$B \models_\Gamma E : S \qquad \Rightarrow \qquad \exists E' \sim E\, [B \vdash_\Gamma E' : S].$$

The proof will occupy the remainder of this section.

9.10. DEFINITION. Let g be a graph.

(i) The *free basis* (of g) (notation B^{F}) is the set

$$B^{\mathrm{F}} = \{n : [\mathcal{U}(n)]_\Gamma^{\mathrm{rc}_g(n)} \mid n \in \mathrm{FV}(g)\}.$$

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

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

9.11. LEMMA. *For all $n \in \mathrm{V}(g)$ one has*

(i) $B^{\mathrm{F}}, B^{\mathrm{B}} \vdash_\Gamma E_n{}^* : \mathcal{U}(n)$;

(ii) $B^{\mathrm{F}}, B^{\mathrm{B}} \vdash_\Gamma \overline{n} : [\mathcal{U}(n)]_\Gamma^{\mathrm{rc}_g(n)}$.

PROOF. By course-of-value 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 $\mathcal{F}, \mathcal{A} \vdash \mathbf{S} : \vec{S} \rightsquigarrow T \mid \Gamma$ with $\mathcal{U}(n) = T$ and $[\mathcal{U}(n_i)]_\Gamma^{\mathrm{rc}(n_i)} \leq S_i$ for all $i \leq k$.

Claim.

$$B^{\mathrm{F}}, B^{\mathrm{B}} \vdash_\Gamma \overline{n} : S_i.$$

Then we are done: we can complete the derivation using the rule *application* and *contraction*. The latter deals with multiple occurrences of the variables in $B^{\mathrm{F}}, B^{\mathrm{B}}$; note that $[\cdot]_\Gamma$ is idempotent, i.e. $[[S]_\Gamma]_\Gamma = [S]_\Gamma$ 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 I.H. (ii) (note that $\deg(n_i) < d$) we have

$$B^{\mathrm{F}}, B^{\mathrm{B}} \vdash_{\Gamma} \overline{n_i} : \mathcal{U}(n_i).$$

Now we can apply *subsumption*.

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

$$B^{\mathrm{F}}, B^{\mathrm{B}} \vdash_{\Gamma} \overline{n_i} : [\mathcal{U}(n_i)]_{\Gamma},$$

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

Case 3. $n_i \in \mathrm{FV}(g)$. Then $B^{\mathrm{F}} \vdash_{\Gamma} n_i : [\mathcal{U}(n_i)]_{\Gamma}^{rc(n_i)}$. Again, by *subsumption* the result follows. $\Box_{\mathit{Claim}}$

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

9.12. Proposition. *Let g be a graph. Furthermore, let $\langle \mathcal{U}, \Gamma \rangle$ be a typing for g^+. Then*

$$B^{\mathrm{F}} \vdash_{\Gamma} E_g : \mathcal{U}(g^+).$$

Proof. Say r, r^+ are the roots of g, g^+ respectively. Note that $r^+ = \mathbf{Root}\ r$, and moreover $\overline{r^+} = \mathbf{Root}\ \overline{r}$ (since $\mathrm{rc}_{g^+}(r^+) = \odot$). Using $\mathcal{F}, \mathcal{A} \vdash \mathbf{Root} : \mathcal{U}(g^+) \rightsquigarrow \mathcal{U}(g^+)$ it follows that $\Gamma \vdash [\mathcal{U}(r)]_{\Gamma}^{\mathrm{rc}_{g^+}(r)} \leq \mathcal{U}(g^+)$. Now, by Lemma 9.11, using the rule *cycle* and the rule *weakening* (also to deal with multiple occurrences of the variables in B^{F}) the result follows. $\Box$

Proof (of Theorem 9.9). Without loss of generality, we can assume that B contains no declarations for variables not appearing in E. Suppose $B \models_{\Gamma} E : S$, say with $\mathcal{U}$ such that $\mathcal{U} \models_{\Gamma} B$ and $\mathcal{U}([\![E]\!]^+) = S$. Observe that $B = B^{\mathrm{F}}_{g,\langle \mathcal{U},\Gamma\rangle}$. Set $E' = E_{[\![E]\!]}$. Then we are done by Proposition 9.12. $\Box$

Subject reduction

Uniqueness typing of graph rewrite rules is defined as follows (cf. Barendsen and Smetsers (1993a))

9.13. Definition. (i) A uniqueness type assignment $\mathcal{U}$ to variables can be extended to patterns (respecting Γ) in the following way:

$$\mathcal{F}, \mathcal{A} \vdash \mathbf{C} : \vec{S} \rightsquigarrow T \mid \Gamma,\ \mathcal{U}(\vec{p}) = \vec{S} \mid \Gamma \ \Rightarrow\ \mathcal{T}(\mathbf{C}\ \vec{p}) = T \mid \Gamma.$$

(ii) The rewrite rule $\mathbf{F}\vec{p} \rightarrow g$ is *(uniqueness) type correct* in $\mathcal{F}$ (containing $\mathbf{F}{:}\vec{S}{\rightsquigarrow}T|\Gamma$) if for some uniqueness typing $\langle \mathcal{U}, \Gamma' \rangle$ for g one has

$$\begin{aligned} \mathcal{U}(\vec{p}) &= \vec{S} \mid \Gamma, \\ \mathcal{U}(g) &= S, \\ \Gamma' \restriction (\vec{S}, T) &= \Gamma \restriction (\vec{S}, T). \end{aligned}$$

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

The following result relates uniqueness typability of function definitions to typability of their graph interpretations.

9.14. LEMMA. $\mathbb{F}$ *is uniqueness type correct* $\Rightarrow$ $[\![\mathbb{F}]\!]$ *is uniqueness type correct.*

9.15. DEFINITION. Let g be a closed graph. g is *typable with S in* Γ (notation $g : \sigma \,|\, \Gamma$) if there exists a uniqueness typing $\langle \mathcal{U}, \Gamma' \rangle$ for g with $\mathcal{U}(g) = S$ and $\Gamma' \restriction S = \Gamma \restriction S$.

SUBJECT REDUCTION THEOREM. *For any closed graph g one has*

$$\left.\begin{array}{r} g : S \,|\, \Gamma, \\ \mathcal{R} \textit{ is uniqueness type correct}, \\ g \twoheadrightarrow_{\mathcal{R}} g' \end{array}\right\} \quad \Rightarrow \quad g' : S \,|\, \Gamma.$$

PROOF. See Barendsen and Smetsers (1993a). $\square$

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.

9.16. THEOREM.

$$\left.\begin{array}{r} B \vdash E : S \,|\, \Gamma \\ \mathbb{F} \textit{ is type correct}, \\ [\![E]\!] \twoheadrightarrow_{[\![\mathbb{F}]\!]} g \end{array}\right\} \quad \Rightarrow \quad B \vdash E_g : S \,|\, \Gamma.$$

10. 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, a 'principal' solution is computable.

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.

The resulting system looks as follows.

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

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

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

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

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

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

10.1. Theorem. *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 $\odot$, $\otimes$.

10.2. Definition. (i) A *type inequality statement* is an expression of the form $S \leq T$, where $S, T \in \widehat{\mathbb{T}}$.

(ii) An *attribute inequality statement* is an expression of the form $u \leq v$, where $u, v \in \mathbb{U}$.

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

10.3. Definition. 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. A *solution* for such a system consists of a uniqueness type substitution $*$ (such that $\ulcorner *(\alpha^u) \urcorner = u$ for each α^u in $\mathcal{S}$) and an attribute environment Γ such that

$$\Gamma \vdash \mathcal{S}^*, \qquad \Gamma \vdash \Delta.$$

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

10.4. Definition. Let B be a uniqueness basis, E a marked expression and $S \in \widehat{\mathbb{T}}$. A system of uniqueness requirements $\mathcal{U}$ is called *uniqueness exact* for B, E, S if for all $*, \Gamma$

(1)

$$\left.\begin{array}{r} *, \Gamma \models \mathcal{U} \\ \Gamma \text{ consistent} \end{array}\right\} \Rightarrow \ B^* \ \vdash^{\mathrm{sd}} \ E : S^* \mid \Gamma.$$

(2)

$$B^* \vdash^{\mathrm{sd}} E : S^* \mid \Gamma \;\Rightarrow\; \begin{array}{l} *', \Gamma' \models \mathcal{U} \\ \text{for some } *', \Gamma' \text{ such that} \\ *', \Gamma' \restriction (B,S) = *, \Gamma \restriction (B,S). \end{array}$$

10.5. Definition. (i) For each $\mathcal{S}$, set $|\mathcal{S}| = \{|S| = |T| \mid \text{`}S \le T\text{'} \in \mathcal{S}\}$. Moreover, for $\mathcal{U} = \langle \mathcal{S}, \Delta \rangle$ set $|\mathcal{U}| = |\mathcal{S}|$.

(ii) Let $*$ be a uniqueness substitution (i.e., $\ulcorner *(\alpha^u) \urcorner = u$ for each α^u in the domain of $*$). Then $*$ *conventionally satisfies* $\mathcal{U}$ (notation $* \models_{\mathrm{c}} \mathcal{U}$) if $|*| \models |\mathcal{U}|$.

10.6. Lemma. *Let* $\mathcal{U} = \langle \mathcal{S}, \Delta \rangle$. *Then*

$$\mathcal{U} \textit{ is uniqueness exact for } B, E, S \;\Rightarrow\; |\mathcal{U}| \textit{ is exact for } |B|, E^-, |S|.$$

Proof. Simple. □

10.7. Proposition. *Let E be an expression, B a basis and $S \in \widehat{\mathbb{T}}$. 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.

$$\begin{array}{rcl}
\mathcal{U}(B, x^{\odot}, S) & = & \langle \{B(x) \le S\}, \emptyset \rangle, \\
\mathcal{U}(B, x^{\otimes}, S) & = & \langle \{B(x) \le S\}, \{\ulcorner S \urcorner = \times\} \rangle, \\
\mathcal{U}(B, \mathbf{S}\vec{E}, S) & = & \bigcup_i \mathcal{U}(B, E_i, T_i) \cup \langle \{R \le S\}, \Delta \rangle \\
 & & \text{if } \mathcal{F}, \mathcal{A} \text{ contains } \mathbf{S} : \vec{T} \rightsquigarrow R \mid \Delta, \\
\mathcal{U}(B, \mathsf{let}\ x = E\ \mathsf{in}\ E', S) & = & \mathcal{U}(B, x{:}\alpha^a, E', S) \cup \mathcal{U}(B, E, \alpha^a) \\
 & & \alpha, a \text{ fresh}, \\
\mathcal{U}(B, \mathsf{letrec}\ \vec{x} = \vec{E}\ \mathsf{in}\ E', S) & = & \bigcup_i \mathcal{U}(B, \vec{x}{:}\vec{\alpha^a}, E_i, {\alpha_i}^{a_i}) \cup \mathcal{U}(B, \vec{x}{:}\vec{\alpha^a}, E', S) \\
 & & \vec{\alpha}, \vec{a} \text{ fresh}, \\
\mathcal{U}(B, \mathsf{case}_{\mathsf{T}} E\ \mathsf{of}\ \vec{P} | \vec{E}, S) & = & \mathcal{U}(B, E, (\mathsf{T}\vec{\alpha})^+) \cup \bigcup_i \mathcal{U}(B, \vec{x_i}{:}(\vec{\sigma_i})^+, E_i, S) \cup \langle \emptyset, \mathrm{Prop}_{\mathsf{T}} \rangle, \\
 & & \text{if } \mathcal{A} \text{ contains } \mathsf{T}\vec{\alpha} = \mathbf{C}_1 \vec{\sigma_1} \mid \cdots;
\end{array}$$

Here $(\tau)^+$ stands for the τ-part of a (uniformly) attributed instance of $\mathsf{T}\vec{\alpha} = \mathbf{C}_1\vec{\sigma_1} \mid \cdots$, and $\mathrm{Prop}_{\mathsf{T}}$ denotes the corresponding propagation requirements (see also section 7). Note that such an instance cannot be obtained just by separate instantiation of all $\mathbf{C}_i$-types, for, this would lead to different versions of $\mathsf{T}\vec{\alpha}$.

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

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

10.8. Lemma. *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. $\square$

Because of our treatment of higher-order functions (involving a restriction on the subtype relation w.r.t. variables) there is some asymmetry between $*$ and Γ in solutions of a system $\mathcal{U}$. Therefore, a notion 'most general solution' cannot be formulated. Instead, we prove that it is decidable whether a substitution $*$, conventionally satisfying $\mathcal{U}$, can be extended (with a suitable Γ) to a solution of $\mathcal{U}$. In that case, a 'principal' Γ can be computed.

In view of this, we first consider *conventionally correct* systems of uniqueness requirements, i.e., systems $\langle \mathcal{S}, \Delta \rangle$ such that $|S| = |T|$ for each '$S \leq T$' in $\mathcal{S}$. We use the denotation $\Gamma \models \mathcal{U}$ in the obvious sense.

10.9. PROPOSITION (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*

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

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

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

Clearly, Γ is computable from $\mathcal{U}$. If Γ is consistent (Lemma 10.8), output $attr(\mathcal{U}) = \Gamma$; otherwise $attr(\mathcal{U}) = \mathsf{fail}$. $\square$

In practice, we will first determine a conventional correct solution for $|\mathcal{U}|$, using conventional techniques, and then try to lift this conventional solution to a uniqueness solution. This leaves some freedom: the attribute $\bullet$, for example, can be placed as a type attribute in a uniqueness type, but can also be assigned indirectly to a type by putting a statement of the form $a \leq \bullet$ in the attribute environment.

10.10. DEFINITION. 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 pair $(*, \Gamma)$ is called an *attribution* of $*_0$ satisfying $\mathcal{U}$. Such an attribution is *principal* if for all attributions $(*', \Gamma')$ of $*$ satisfying $\mathcal{U}$ there exists an attribute substitution $\diamond$ such that

$$*' = *^{\diamond}, \qquad \Gamma' \Vdash \Gamma^{\diamond}.$$

PRINCIPAL ATTRIBUTION THEOREM. *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 $(*, \Gamma)$. $\square$

11. Uniqueness Type Inference in Clean

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. Since recursive specifications are allowed, determination of such an environment is undecidable in general. In fact, the *Clean* compiler adopts the Hindley-Milner approach towards recursion: in the definition of, say, $\mathbf{F}$, all occurrences of $\mathbf{F}$ should be typed with $\mathbf{F}$'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 10.7.

This enables us to determine a uniqueness type for $\mathbf{F}$ in the following way. Say $\mathbf{F}$ has the form $\mathbf{F}\vec{x} \to E$. The first step is to build an initial basis B_0 containing

$$x_1 = {\alpha_1}^{a_1}, \cdots, x_k = {\alpha_k}^{a_k}.$$

Set $S_0 = \alpha^a$. As usual, all type and attribute variables are fresh. Define $\mathcal{U}_0 = \mathcal{U}(B_0, E, S_0)$. By the Unification Theorem we can find a (most general) conventional solution for $|\mathcal{U}_0|$. Applying the Principal Attribution Theorem yields an attribution $(*, \Gamma)$. Now the type for $\mathbf{F}$ is obtained as follows. Set $\Gamma_{\mathbf{F}} = \Gamma \restriction (B_0^*, S_0^*)$, then

$$\mathbf{F} : ({\alpha_1}^{a_1})^*, \ldots, ({\alpha_k}^{a_k})^*) \rightsquigarrow S_0^* \mid \Gamma_{\mathbf{F}}.$$

Of course, if one of these steps fails then $\mathbf{F}$ is not typable.

Due to the aforementioned asymmetry between $*$ and Γ, it might be the case that there exists an attributable conventional solution for $\mathcal{U}_0$ despite of the fact that the most general solution itself is not attributable. In *Clean*, however, such a rule is considered to be untypable, so it is not tried to derive a uniqueness type using a more specific conventional solution if the attribution of the most general one fails.

12. 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.

In-line updates

This present work has been inspired by Guzmán and Hudak (1990). This paper addresses the mutability problem using a 'single threaded polymorphic lambda calculus' (*poly*-λ_{st}). Instead of an operational semantics directly based on graph rewriting, they use lambda-graph reduction of Wadsworth (1971). Type reconstruction is (roughly) described by defining a type inference algorithm; correctness of this algorithm is not included.

In Turner et al. (1995), a type system is presented that is strongly related to ours. However, the design was led 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.

Perhaps the most remarkable fact about the systems of both Guzmán and Hudak (1990) and Turner et al. (1995) is that, none of these has been implemented yet, not to mention that they are incorporated in a real programming language.

Uniqueness Typing and Affine Logic

The connection between uniqueness typing and linear/affine logic can be sketched as follows.

The layer of $\bullet$ types corresponds to *affine logic*: inside this layer, objects are treated in an affine way: there is cancellation (weakening) but no duplication (contraction).

The second layer ($\times$) has no limitations on the use of objects: both cancellation and duplication are possible. Indeed, since $[S] = S$ if $\ulcorner S \urcorner = \times$, we have in fact contraction for $\times$ types. This layer corresponds to ordinary (intuitionistic) logic.

We can move from the affine layer to the $\times$ layer (thus giving up linearity properties) in two ways: either voluntarily (along the relation $\leq$) of forced (by the type correction operator $[\cdot]$, in case of multiple use).

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'.

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

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

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

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

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

Benton, P.N. (1994). A mixed linear and non-linear logic: Proofs, terms and models, *Technical Report 352*, Computer Laboratory, University of Cambridge.

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.

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

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.