ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

David N. Turner (Ed)

Functional Programming, Glasgow 1995

Proceedings of the 1995 Glasgow Workshop on Functional
Programming,
Ullapool, Scotland, 10-12 July 1995

Paper:

Uniqueness Typing in Natural Deduction Style

Erik Barendsen and Sjaak Smetse

Published in collaboration with the
British Computer Society

©Copyright in this paper belongs to the author(s)

Uniqueness Typing in Natural Deduction Style

Erik Barendsen Sjaak Smetsers

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

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.

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 arguments with reference count 1.

The uniqueness type system for graph rewrite systems (presented in Barendsen and Smetsers (1993a) and (1993b)) offers the possibility to indicate such reference count 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 : (Char×, File•)  $\rightsquigarrow$  File•`. Here, [•], [×] 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).

The present paper describes a simplified version of the 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).

We start with a specification of the formal language and define a Curry-like (conventional) type system. After a very brief introduction to uniqueness typing, a description of the uniqueness type assignment system is given. For both systems we prove preservation of typing during reduction and the existence of principal types.

The original uniqueness type system is rather complex. To avoid that the reader gets entangled in technical details, the reference analysis is kept as simple as possible: it does not take the evaluation order into account. For details, see Barendsen and Smetsers (1993a).

Independently, Turner et al. (1995) have developed a strongly related system based on λ -calculus. The research reported there has been guided by different objectives.

Uniqueness typing is now part of the functional programming language *Clean*, see Plasmeijer and van Eekelen (1995).

2 Syntax

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

$$\begin{aligned} E &::= x \mid \mathbf{S}(E_1, \dots, E_k) \mid \text{let } x = E \text{ in } E' \mid \text{letrec } \vec{x} = \vec{E} \text{ in } E' \mid \text{case } E \text{ of } \vec{P} \mid \vec{E}, \\ P &::= \mathbf{C}(x_1, \dots, x_k). \end{aligned}$$

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). Patterns (indicated by P in the above syntax) are supposed to be *linear*: no variable is occurring more than once in the same pattern.

The expressions are interpreted as graphs. Sharing manifests itself by multiple occurrences of the same variables. Sharing of compound structures is expressed by a *let* binding, whereas *letrec* introduces cyclic dependencies. E.g., the expression

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

denotes the graph

Operations on objects are defined using *function definitions*. The general form of these is

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

For example,

$$\text{Append}(\ell_1, \ell_2) = \text{case } \ell_1 \text{ of } \begin{array}{ll} \mathbf{Cons}(h, t) & \mid \mathbf{Cons}(h, \text{Append}(t, \ell_2)) \\ \mathbf{Nil} & \mid \ell_2. \end{array}$$

The operational semantics of our language is given by the concept of *Term Graph Rewriting*, see Barendregt et al. (1987). Each expression E is translated to a graph denoted by $\llbracket E \rrbracket$. Function definitions are translated into rewrite rules, inducing a (multistep) rewrite relation $\twoheadrightarrow$ on graphs (cf. Barendsen and Smetsers (1993a)). In contrast with Ariola and Klop (1995), we refrain from defining a reduction relation on the expressions themselves.

3 Conventional Typing

Types are built up from type variables and type constructors.

$$\sigma ::= \alpha \mid \mathbf{T}\vec{\sigma} \mid \sigma_1 \rightarrow \sigma_2.$$

Here, $\mathbf{T}$ ranges over *algebraic type constructors* which are assumed to be introduced by an *algebraic type system* $\mathcal{A}$ containing specifications like

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

The function space type constructor $\rightarrow$ is used when dealing with *higher-order functions*. We will treat these only in the uniqueness case.

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

$$T\vec{\alpha} = C_1\vec{\sigma}_1 | \dots$$

be a declaration in $\mathcal{A}$. Then

$$\mathcal{A} \vdash C_i : \vec{\sigma}_i \rightsquigarrow T\vec{\alpha}$$

For example, for lists one has

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

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

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

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

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

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).

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

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.

$B, x:\sigma \vdash x : \sigma \quad (\text{variable})$
$\frac{\mathcal{F}, \mathcal{A} \vdash \mathbf{S} : \vec{\sigma} \rightsquigarrow \tau \quad B \vdash \vec{E} : \vec{\sigma}}{B \vdash \mathbf{S}\vec{E} : \tau} \quad (\text{application})$
$\frac{B \vdash E : \sigma \quad B, x:\sigma \vdash E' : \tau}{B \vdash \text{let } x = E \text{ in } E' : \tau} \quad (\text{sharing})$
$\frac{B, \vec{x}:\vec{\sigma} \vdash E_i : \sigma_i \quad B, \vec{x}:\vec{\sigma} \vdash E' : \tau}{B \vdash \text{letrec } \vec{x} = \vec{E} \text{ in } E' : \tau} \quad (\text{cycle})$
$\frac{B \vdash E : \tau \quad \mathcal{A} \vdash \mathbf{C}_i : \vec{\sigma}_i \rightsquigarrow \tau \quad B, \vec{x}_i:\vec{\sigma}_i \vdash E_i : \tau'}{B \vdash \text{case } E \text{ of } \vec{P} \vec{E} : \tau' \quad (\text{if } P_i = \mathbf{C}_i \vec{x}_i)} \quad (\text{pattern matching})$

This concludes the treatment of expressions. As to function definitions, the environment $\mathcal{F}$ should be consistent with these. We say that the function

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

say with standard type $\mathbf{F} : \vec{\sigma} \rightsquigarrow \tau$, is *type correct* if

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

Typing is preserved during reduction (the so-called *subject reduction property*).

SUBJECT REDUCTION THEOREM. *Suppose the function definitions are type correct. Then*

$$B \vdash E : \sigma, \llbracket E \rrbracket \rightarrow g \quad \Rightarrow \quad \exists E' \llbracket E' \rrbracket = g \ \& \ B \vdash E' : \sigma.$$

The system has the *principal type property*: each typable expression E has a type from which all other types for E can be obtained by instantiation.

PRINCIPAL TYPE THEOREM. *Let E be typable. Then there exist B_0, σ_0 (computable from E) such that $B_0 \vdash E : \sigma_0$ and for any B and σ*

$$B \vdash E : \sigma \quad \Rightarrow \quad B \supseteq B_0^*, \sigma = \sigma_0^* \text{ for some substitution } *.$$

4 Simple Uniqueness Typing

Uniqueness typing combines conventional typing and linear typing, through a reference count administration. An environment type $\mathbf{F} : \sigma^\bullet \rightsquigarrow \dots$ 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 $\mathbf{G} : \dots \rightsquigarrow \sigma^\bullet$, 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 $F : \sigma^\times \rightsquigarrow \dots$ then still $\mathbf{F}(\mathbf{G}(E))$ is type correct. This is expressed in a subtype relation, 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.

The non-unique ('conventional') and unique ('linear') types are also connected by a correction mechanism: 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 arguments via *data paths* instead of a single reference. This gives rise to 'hidden sharing' of objects by access via intermediate data nodes. For example, if a function $\mathbf{F}$ has access to a list with non-unique spine, the list elements should also be considered as non-unique for $\mathbf{F}$: 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. This *uniqueness propagation* can be expressed using the $\leq$ relation. In the case of lists, for example,

$$\mathbf{Cons} : (\alpha^u, \text{List}^v(\alpha^u)) \rightsquigarrow \text{List}^v(\alpha^u)$$

is well-attributed if $v \leq u$ (this indeed excludes a constructor for $\text{List}^\times(\text{Int}^\bullet)$). Note that the attribute of a non-variable type is attached to its topmost type constructor.

In Barendsen and Smetsers (1993a), the typing system is shown to be sound with respect to term graph rewriting, with the above-mentioned interpretation of uniqueness.

Algebraic Uniqueness Types

We will first describe the system without the type constructor $\rightarrow$. Below, $S, T, \dots$ 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,

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

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 :

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

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

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

The notion of standard type is adapted in the following way. As before, standard types of function symbols ($F : \vec{S} \rightsquigarrow T$) are collected in an environment $\mathcal{F}$.

As can be seen from the List example, there are several standard types for each data constructor. Say the algebraic environment $\mathcal{A}$ contains

$$\mathsf{T}\vec{\alpha} = \mathsf{C}_1\vec{\sigma}_1 \mid \dots$$

A set of standard types for C_i consists of attributed versions of the conventional type $\vec{\sigma}_i \rightsquigarrow \mathsf{T}\vec{\alpha}$, 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 (1993a) 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 $\vec{S} \rightsquigarrow T$ we set $\mathcal{A} \vdash \mathsf{C} : \vec{S} \rightsquigarrow T$ as before.

Symbol types are instantiated via the rule

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

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.

$$\boxed{\begin{array}{c} x:S \vdash x : S \quad \text{(variable)} \\[10pt] \frac{\mathcal{F}, \mathcal{A} \vdash \mathsf{S} : \vec{S} \rightsquigarrow T \quad B_i \vdash E_i : S_i}{\vec{B} \vdash \mathsf{S}\vec{E} : T} \quad \text{(application)} \\[10pt] \frac{B \vdash E : S \quad B', x:S \vdash E' : T}{B, B' \vdash \text{let } x = E \text{ in } E' : T} \quad \text{(sharing)} \\[10pt] \frac{B_i, \vec{x}:[\vec{S}] \vdash E_i : S_i \quad B', \vec{x}:[\vec{S}] \vdash E' : T}{\vec{B}, B' \vdash \text{letrec } \vec{x} = \vec{E} \text{ in } E' : T} \quad \text{(cycle)} \\[10pt] \frac{B \vdash E : T \quad \mathcal{F}, \mathcal{A} \vdash \mathsf{C}_i : \vec{S} \rightsquigarrow T \quad B', \vec{x}_i : \vec{S}_i \vdash E_i : T'}{B, B' \vdash \text{case } E \text{ of } \vec{P} | \vec{E} : T' \quad (\text{if } P_i = \mathsf{C}_i \vec{x}_i)} \quad \text{(pattern matching)} \end{array}}$$

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

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.

$$\boxed{\begin{array}{c} \frac{B \vdash E : T}{B, x:S \vdash E : T} \text{ (weakening)} \\[1em] \frac{B, y:[S], z:[S] \vdash E : T}{B, x:S \vdash E[y := x, z := x] : T} \text{ (contraction)} \end{array}}$$

Like in the conventional case, $\mathcal{F}$ should be consistent with the function definitions. A function

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

say with standard type $\mathbf{F} : \vec{S} \rightsquigarrow T$, 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

$$\lambda \vec{x}. \mathbf{F}(E_1, \dots, E_k, x_{k+1}, \dots, x_{\text{arity}(\mathbf{F})}).$$

If these partial applications contain unique subexpressions one has to be careful. Consider, for example, a function $\mathbf{F}$ with type $\mathbf{F} : (\sigma^\bullet, \tau^\times) \rightsquigarrow \sigma^\bullet$ in the 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 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 $\triangle$, for any unique object that does not coerce to a non-unique variants. This has been described in Barendsen and Smetsers (1993a). An alternative solution is the administration of so-called *regions* introduced by Reynolds (1995).

Instead of introducing a new attribute, the present paper considers the $\rightarrow$ constructor in combination with the $\bullet$ attribute as special: it is not permitted to discard its uniqueness. This 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 $\rightarrow$ 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 $\text{sign}(\rightarrow) = (\ominus, \oplus)$. The subtyping relation $\leq$ is defined by induction.

Notation. $S \leq^{\oplus} T \Leftrightarrow S \leq T$, $S \leq^{\ominus} T \Leftrightarrow T \leq S$,
 $S \leq^{\top} T \Leftrightarrow S \leq^{\oplus} T$ and $S \leq^{\ominus} T$,
 $\vec{S} \leq^{\vec{s}} \vec{T} \Leftrightarrow S_i \leq^{s_i} T_i$ for each i .

Now set

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

E.g.

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

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

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

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

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

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

$$\boxed{\frac{\mathcal{F}, \mathcal{A} \vdash \mathbf{F}_{k+1} : (\vec{S}, T) \rightsquigarrow T' \quad u \leq \Pi^{\top} \vec{S}^{\top}}{\mathcal{F}, \mathcal{A} \vdash \mathbf{F}_k : \vec{S} \rightsquigarrow (T \xrightarrow{u} T')}} \text{ (Curry)}$$

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.

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

5 Polymorphic Uniqueness Typing

In order to denote uniqueness schemes, we extend the attribute set with *attribute variables* ($a, b, a_1, \dots$). This increases the expressivity 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 *coercion environments*. For example, the standard type of the symbol **Cons** is now expressed by

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

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

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

$$\begin{aligned} \Gamma \vdash u \leq v & \quad \text{if } (u \leq v) \in \Gamma, \\ \Gamma \vdash u \leq u, \quad \Gamma \vdash u \leq \times, \quad \Gamma \vdash \bullet \leq u \end{aligned}$$

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 $(u \leq v) \in \Gamma'$. We say that Γ is *consistent* if $\Gamma \not\vdash \times \leq \bullet$.

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

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

The context rules become

$$\boxed{\begin{aligned} & \frac{\mathcal{F}, \mathcal{A} \vdash \mathbf{S} : \vec{S} \rightsquigarrow T \mid \Gamma \quad \Gamma' \vdash \Gamma}{\mathcal{F}, \mathcal{A} \vdash \mathbf{S} : \vec{S} \rightsquigarrow T \mid \Gamma'} \text{ (attribute instantiation)} \\ & \frac{\mathcal{F}, \mathcal{A} \vdash \mathbf{S} : \vec{S} \rightsquigarrow T \mid \Gamma \quad \Gamma \vdash \ulcorner \alpha \urcorner = \ulcorner R \urcorner}{\mathcal{F}, \mathcal{A} \vdash \mathbf{S} : \vec{S}[\alpha := R] \rightsquigarrow T[\alpha := R] \mid \Gamma} \text{ (instantiation)} \\ & \frac{\mathcal{F}, \mathcal{A} \vdash \mathbf{F}_{i+1} : (\vec{S}, T) \rightsquigarrow T' \mid \Gamma \quad \Gamma \vdash u \leq \ulcorner \vec{S} \urcorner}{\mathcal{F}, \mathcal{A} \vdash \mathbf{F}_i : \vec{S} \rightsquigarrow (T \xrightarrow{u} T') \mid \Gamma} \text{ (Curry)} \end{aligned}}$$

The correction operation is also relativized: for example, $[S \xrightarrow{u} T]_{\Gamma} = S \xrightarrow{\times} T$ if $\Gamma \vdash u = \times$.

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

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

and the subsumption rule

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

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. By $B \vdash E : S \mid \Gamma$ we denote that $B \vdash_{\Gamma'} E : S$ is derivable for some ‘conservative’ extension Γ' of Γ (with respect to attributes occurring in Γ , the environment Γ' proves the same statements as the original Γ). Alternatively, one could extend the deduction system with rules for explicit simplification of environments.

Type correctness of function definitions can now be expressed as follows. The function

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

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

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

The subject reduction property also holds for the uniqueness type system, cf. Barendsen and Smetsers (1993a).

We are interested in attributed versions of conventional type derivations: given B, E and σ and a derivation showing $B \vdash E : \sigma$, we wish to construct a uniqueness variant (in the obvious sense: assigning the same underlying conventional types to the subexpressions of E) of this derivation, yielding B_0, S_0, Γ_0 with $|B_0| = B, |S_0| = \sigma$ and $B_0 \vdash E : S_0 \mid \Gamma_0$.

PRINCIPAL ATTRIBUTION THEOREM. *Suppose a given derivation of $B \vdash E : \sigma$ is attributable. Then there exists an attribution, B_0, S_0, Γ_0 (computable from the conventional derivation, B, E and σ) such that for any other attribution, B_1, S_1, Γ_1 ,*

$$B_1 = B_0^\diamond, \quad S_1 = S_0^\diamond, \quad \Gamma_1 \vdash \Gamma_0^\diamond$$

for some attribute substitution $\diamond$.

This decidability result (in the original graph framework) has been addressed in Barendsen and Smetsers (1995).

Example

The (higher-order) function **Map** is defined as usual:

$$\begin{aligned} \mathbf{Map}(f, \ell) &= \text{case } \ell \text{ of } \begin{array}{l|l} \mathbf{Cons}(h, t) & \mathbf{Cons}(\mathbf{Ap}(f, h), \mathbf{Map}(f, t)) \\ \mathbf{Nil} & \mathbf{Nil} \end{array} \\ &= E(f, \ell), \text{ say.} \end{aligned}$$

Then

$$h:\alpha^a, f:\alpha^a \xrightarrow{d} \beta^b, t:\text{List}^e(\alpha^a) \vdash \mathbf{Cons}(\mathbf{Ap}(f, h), \mathbf{Map}(f, t)) : \text{List}^e(\beta^b) \mid \times \leq d. \quad (*)$$

Moreover

$$f:\alpha^a \xrightarrow{d} \beta^b, \ell:\text{List}^e(\alpha^a) \vdash E(f, \ell) : \text{List}^e(\beta^b) \mid \times \leq d.$$

This also validates $(\alpha^a \xrightarrow{*} \beta^b, \text{List}^e(\alpha^a)) \rightsquigarrow \text{List}^e(\beta^b)$ as standard type for **Map** used in $(*)$.

6 Concluding Remarks

We have presented *Clean*'s uniqueness typing system in natural deduction style. The original systems have been shown to be decidable in the sense that principal types can be determined effectively, see Barendsen and Smetsers (1993a) and (1995). The present framework not only provides more direct proofs of these results, but also fits closely to more common methods based on substructural logics. The relation with the approach of Benton (1994) is subject to further research.

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.
- 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 (1993a). Conventional and uniqueness typing in graph rewrite systems, *Technical Report CSI-R9328*, Computing Science Institute, University of Nijmegen. See also Barendsen (1995).
- 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 (1995). 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.
- 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.
- 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.
- 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.