To appear in: proceedings of SEGRAGRA'95, Elsevier.

A Derivation System for Uniqueness Typing

Extended abstract

Erik Barendsen Sjaak Smetsers
University of Nijmegen*

1. Introduction

Traditional functional programming languages are unable to deal with operations with side effects. Indeed, by admitting these operations (such as file manipulations) one risks the loss of referential transparency since these involve changing the state of an input object. In recent years, various proposals have been brought up as solutions to this shortcoming (e.g. Wadler (1990), Guzmán and Hudak (1990)). The essence of these solutions is the restriction of destructive operations to input objects that occur only once.

The uniqueness type system for graph rewrite systems (as presented in Barendsen and Smetsers (1993a) and (1993b)) offers the possibility to indicate 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** : $(\mathsf{Char}^{\times}, \mathsf{File}^{\bullet}) \rightarrowtail \mathsf{File}^{\bullet}$. Here, $\bullet, \times$ stand for 'unique' and 'non-unique' respectively. Thus uniqueness typing can be regarded as a combined linear (Wadler (1990)) and conventional type system. A logical/categorical approach appears in Benton (1994). In our system, unique types are connected with their non-unique variants via a subtyping mechanism. The uniqueness type system is now part of the functional programming language *Clean*.

Originally, the notion of typing for graphs and rewrite rules had not been defined inductively but instead specified in terms of local requirements for a type assignment to nodes. The present paper describes a simplified version of our original system, using an inductive syntax and natural deduction style type assignment system. This captures the core of uniqueness typing of graphs, and makes the relation with linear logic more visible. Apart from this, it makes the work on uniqueness more accessible, especially for people not familiar with graph rewriting. The object language is similar to the equational approach of Term Graph Rewriting as introduced by Ariola and Klop (1993).

We start with a specification of the formal language and define a Curry-like (conventional) type system for it. After a very brief introduction to uniqueness typing, a

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

description of the uniqueness type assignment system is given. For both systems we prove preservance 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 we have left out some of the refinements. For instance, we do not deal with higher-order functions and the reference analysis is kept as simple as possible: it does not take the evaluation order into account.

2. 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.

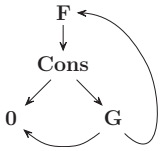
$$E \ ::= \ x \mid \mathbf{F}(E_1, \ldots, E_k) \mid \mathsf{let}\ x = E_1\ \mathsf{in}\ E_2 \mid \mu x[E].$$

Here x ranges over term variables, and $\mathbf{F}$ over some set of *symbols* of fixed arity (with intended interpretation functions and data constructors). The set of *free variables* of an expression, $\mathrm{FV}(E)$, is defined as usual. A term is called *algebraic* if it is built from data constructors and variables only, using application.

Ordinary sharing can be expressed using a let construct, whereas μ introduces (direct) cyclic dependencies. E.g., the expression

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

denotes the graph



Operations on terms are defined using *rewrite rules*. The general form of such a rule is

$$\mathbf{F}(A_1, \ldots, A_k) \to E,$$

where $\mathrm{FV}(E) \subseteq \mathrm{FV}(\vec{A})$, and the A_i are algebraic expressions. We only consider *left-linear* rewrite rules, i.e., rules in which each variable occurs only once in the left-hand side. An example is the rule

$$\mathbf{Append}(\mathbf{Cons}(h, t), l) \to \mathbf{Cons}(h, \mathbf{Append}(t, l)).$$

The rewrite semantics of expressions (according to some set $\mathcal{R}$ of rules) is defined as usual. By $E \underset{\mathcal{R}}{\twoheadrightarrow} E'$ we denote that E rewrites to E' in zero or more steps.

3. Conventional Typing

Types are built up from type variables and type constructors.

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

A standard type constructor is $\rightarrow$; the others are *algebraic type constructors* which are assumed to be introduced by algebraic specifications like

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

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

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

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

$$\mathbf{Nil} : \mathsf{List}(\alpha), \qquad \mathbf{Cons} : (\alpha, \mathsf{List}(\alpha)) \rightarrowtail \mathsf{List}(\alpha).$$

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

Our system deals with *typing statements* of the form

$$B \ \vdash_{\mathcal{E}} \ E : \sigma,$$

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.

$$B, x{:}\sigma \ \vdash_{\mathcal{E}} \ x : \sigma \quad \text{(variable)} \qquad \frac{B \ \vdash_{\mathcal{E}} \ \vec{E} : \vec{\sigma} \qquad \mathcal{E} \vdash \mathbf{F} : \vec{\sigma} \rightarrowtail \tau}{B \ \vdash_{\mathcal{E}} \ \mathbf{F}\vec{E} : \tau} \text{(application)}$$

$$\frac{B \ \vdash_{\mathcal{E}} \ E_1 : \sigma_1 \qquad B, x{:}\sigma_1 \ \vdash_{\mathcal{E}} \ E_2 : \sigma_2}{B \ \vdash_{\mathcal{E}} \ \mathsf{let}\ x = E_1\ \mathsf{in}\ E_2 : \sigma_2} \text{(sharing)} \qquad \frac{B, x{:}\sigma \ \vdash_{\mathcal{E}} \ E : \sigma}{B \ \vdash_{\mathcal{E}} \ \mu x[E] : \sigma} \text{(cycle)}$$

Instantiation of environment types is defined by the following two rules ($[\alpha := \rho]$ denotes substitution).

$$\mathcal{E}, \mathbf{F}{:}\vec{\sigma} \rightarrowtail \tau \vdash \mathbf{F} : \vec{\sigma} \rightarrowtail \tau \qquad \frac{\mathcal{E} \vdash \mathbf{F} : \vec{\sigma} \rightarrowtail \tau}{\mathcal{E} \vdash \mathbf{F} : \vec{\sigma}[\alpha := \rho] \rightarrowtail \tau[\alpha := \rho]}$$

This concludes the treatment of graph typings. If we consider objects in the context of rewrite rules, the symbol environment $\mathcal{E}$ in question should be consistent with these rules. We say that $\mathcal{E}$ is $\mathcal{R}$-OK if for each rule

$$\mathbf{F}(A_1, \ldots, A_k) \rightarrow E,$$

say with $\mathcal{E}$ containing $\mathbf{F}{:}(\sigma_1, \ldots, \sigma_k) \rightarrowtail \tau$ one has for some B

$$B \ \vdash_{\mathcal{E}} \ A_1 : \sigma_1, \quad \ldots, \quad B \ \vdash_{\mathcal{E}} \ A_k : \sigma_k, \quad B \ \vdash_{\mathcal{E}} \ E : \tau.$$

We show that typing is preserved during reduction (the so-called *subject reduction property*).

THEOREM.

$$\left.\begin{array}{r} B \vdash_{\mathcal{E}} E : \sigma \\ E \underset{\mathcal{R}}{\twoheadrightarrow} E' \\ \mathcal{E} \text{ is } \mathcal{R}\text{-OK} \end{array}\right\} \Rightarrow B \vdash_{\mathcal{E}} E' : \sigma.$$

Furthermore, we prove that the systems has the *principal type property*: each typable expression E has a type from which all other type for E can be obtained by instantiation.

THEOREM. *Suppose E is typable. Then there exist B_0, σ_0 such that for any B and σ*

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

4. Uniqueness Typing

Uniqueness typing combines conventional typing and linear typing, through a reference count administration. An environment type $\mathbf{F} : \sigma^\bullet \rightarrowtail \cdots$ 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 $\mathbf{G} : \cdots \rightarrowtail \sigma^\bullet$, 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 $F : \sigma^\times \rightarrowtail \cdots$ 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.

Pattern matching 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 'deeper access' is taken into account by a restriction on the uniqueness types of data constructors. In the case of lists, e.g., one can distinguish uniqueness of the spine and uniqueness of the elements. If a function $\mathbf{F}$ has access to a list with non-unique spine, it cannot be expected that the list elements are unique for $\mathbf{F}$: other functions may access them via the spine. Thus the type $\mathsf{List}^\times(\mathsf{Int}^\bullet)$ does not make sense. Uniqueness of the element type should 'propagate' to uniqueness of the list type. This can be expressed using the $\leq$ relation: $\mathsf{List}^a(\mathsf{Int}^b)$ is correct iff $a \leq b$. Note that the variant $\mathsf{List}^\times(\mathsf{Int}^\bullet)$ is indeed excluded since $\times \not\leq \bullet$. For the function type constructor $\rightarrow$ there are no propagation assumptions (it has no standard term constructors).

In Barendsen and Smetsers (1993a), we described a currying mechanism for functions (involving a restriction on the subtyping relation w.r.t. $\rightarrow$ types). In this abstract we refrain from going into this.

Apart from the subtype relation, the non-unique ('conventional') and unique ('linear') types are connected by a correction mechanism: a unique result may be used more than once, as long as only non-unique supertypes are required. Cyclic objects (with their inherent sharing) are treated separately by requiring the result type to be non-unique.

We now give a more technical account. In order to denote uniqueness schemes, we replace the concrete attributes with attribute labels ($a, b, a_1, \ldots$). Below, $S, T, \ldots$ range over labelled types. The outermost attribute of S is denoted by $\ulcorner S \urcorner$.

Uniqueness constraints are indicated by (finite) sets of label inequalities. These can be used to express the uniqueness propagation mentioned above. For example,

$$\mathsf{List}^a(\mathsf{Int}^b) \mid a \leq b$$

denotes the variants $\mathsf{List}^\bullet(\mathsf{Int}^\bullet)$, $\mathsf{List}^\bullet(\mathsf{Int}^\times)$ and $\mathsf{List}^\times(\mathsf{Int}^\times)$. Also attribute equalities can be expressed: by $a = b$ we abbreviate $a \leq b$, $b \leq a$. Concrete assignment can be done by $a = \bullet$ or $a = \times$ respectively. E.g., adding $b = \bullet$ in the above example reduces the variants to $\mathsf{List}^\bullet(\mathsf{Int}^\bullet)$.

Let Γ be a set of uniqueness constraints. We say that $u \leq v$ is *derivable* from Γ (notation $\Gamma \vdash u \leq v$) if $\Gamma \vdash u \leq v$ can be produced by the axioms

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

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 $(u \leq v) \in \Gamma'$. We say that Γ is *consistent* if $\Gamma \nvdash \times \leq \bullet$.

The relation $\leq$ is extended to types: the validity of $S \leq S'$ in Γ depends subtypewise on the validity of $\Gamma \vdash a \leq a'$ and/or $\Gamma \vdash a' \leq a$, with a, a' attributes in S, S'. As usual, the directions of these positionwise inequalities depends on the covariance and contravariance of the type constructors. One has, for example,

$$\Gamma \vdash \mathsf{List}^a(\mathsf{Int}^b \xrightarrow{c} \mathsf{Bool}^d) \leq \mathsf{List}^{a'}(\mathsf{Int}^{b'} \xrightarrow{c'} \mathsf{Bool}^{d'}) \quad \text{iff} \quad \Gamma \vdash a \leq a', c \leq c', b' \leq b, d \leq d'.$$

By $\mathrm{Con}(S)$ we denote the set of uniqueness constraints expressing propagation requirements (so-called internal consistency). For example (remember that $\to$ has no propagation requirements),

$$\mathrm{Con}(\ \mathsf{List}^a(\mathsf{Int}^b \xrightarrow{c} \mathsf{List}^d(\mathsf{Char}^e))\ ) = \{a \leq c, d \leq e\}.$$

In the typing system, we consider constraints which at least imply the internal consistency of the types in question. This will be made precise in the type assignment rules.

In order to account for multiple references to the same object we introduce a *uniqueness correction*: if an object has type S, then only non-unique versions of S may be used. Given S and Γ, we construct the minimal non-unique supertype (w.r.t. $\leq$) $\widehat{S}$ with constraints $\widehat{\Gamma}$. For $\mathsf{List}^a(\mathsf{Int}^b \xrightarrow{c} \mathsf{Bool}^d)$ this gives

$$\mathsf{List}^{\hat{a}}(\mathsf{Int}^b \xrightarrow{\hat{c}} \mathsf{Bool}^d) \quad \mid \quad \widehat{a} = \times, \widehat{c} = \times.$$

A *uniqueness typing statement* has the form

$$B \mid \Gamma \ \vdash_{\mathcal{E}} \ E : S \mid \Delta$$

where Γ, Δ contain uniqueness constraints for the types in B and S respectively. The environment $\mathcal{E}$ now contains declarations

$$F : \vec{S} | \Gamma \rightarrowtail T | \Delta,$$

where Γ, Δ are consistent attribute environments on the input types $\vec{S}$ and output type T, with $\Gamma \vdash \mathrm{Con}(\vec{S})$ and $\Delta \vdash \mathrm{Con}(T)$.

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. Moreover, all (combinations of) attribute environments are implicitly required to be consistent.

The rules for type assignment are the following.

$$\frac{\Gamma \vdash \mathrm{Con}(S)}{x{:}S \mid \Gamma \;\vdash_{\mathcal{E}}\; x : S \mid \Gamma} \text{(variable)}$$

$$\frac{B_i \mid \Gamma_i \;\vdash_{\mathcal{E}}\; E_i : S_i \mid \Delta_i \qquad \mathcal{E} \vdash \mathbf{F} : \vec{S'} | \vec{\Delta} \rightarrowtail T | \Delta' \qquad \Delta_i \vdash S_i \leq S_i'}{\vec{B} \mid \vec{\Gamma} \;\vdash_{\mathcal{E}}\; \mathbf{F}\vec{E} : T \mid \Delta'} \text{(application)}$$

$$\frac{B_1 \mid \Gamma_1 \;\vdash_{\mathcal{E}}\; E_1 : S \mid \Delta_1 \qquad B_2, x{:}S \mid \Gamma_2, \Delta_1 \;\vdash_{\mathcal{E}}\; E_2 : T \mid \Delta_2}{B_1, B_2 \mid \Gamma_1, \Gamma_2 \;\vdash_{\mathcal{E}}\; \mathsf{let}\ x = E_1\ \mathsf{in}\ E_2 : T \mid \Delta_2} \text{(sharing)}$$

$$\frac{B, x{:}S \mid \Gamma \;\vdash_{\mathcal{E}}\; E : S \mid \Delta \qquad \Gamma \vdash \ulcorner S \urcorner = \times}{B \mid \Gamma \;\vdash_{\mathcal{E}}\; \mu x[E] : S \mid \Delta} \text{(cycle)}$$

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.

$$\frac{B \mid \Gamma \;\vdash_{\mathcal{E}}\; E : T \mid \Delta \qquad \Gamma, \Gamma' \vdash \mathrm{Con}(S)}{B, x{:}S \mid \Gamma, \Gamma' \;\vdash_{\mathcal{E}}\; E : T \mid \Delta} \text{(weakening)}$$

$$\frac{B, y{:}\widehat{S}, z{:}\widehat{S} \mid \Gamma, \widehat{\Gamma} \;\vdash_{\mathcal{E}}\; E : T \mid \Delta \qquad \Gamma \vdash \mathrm{Con}(S)}{B, x{:}S \mid \Gamma \;\vdash_{\mathcal{E}}\; E[y := x, z := x] : T \mid \Delta} \text{(contraction)}$$

Instantiation of environment types proceeds in two steps: first, one may use a more restrictive attribute environment than the one specified in $\mathcal{E}$. Second, the type variables may be replaced by concrete uniqueness types (with the same attribute) by substitutions $[\alpha := U|\Delta]$, with $\Delta \vdash \mathrm{Con}(U)$. (We assume that type variables are

'uniformly labelled' throughout an environment type.)

$$\frac{\Gamma' \vdash \Gamma}{\mathcal{E}, \mathbf{F}{:}\vec{S}|\Gamma \rightarrowtail T|\Delta \vdash \mathbf{F} : \vec{S}|\Gamma' \rightarrowtail T|\Gamma', Delta}$$

$$\frac{\mathcal{E} \vdash \mathbf{F} : \vec{S}|\Gamma \rightarrowtail T|\Gamma' \qquad [\alpha := U|\Delta] \text{ substitution}}{\mathcal{E} \vdash \mathbf{F} : \vec{S}[\alpha := U]|\Gamma, \Delta \rightarrowtail T[\alpha := U]|\Gamma', \Delta}$$

(In applications of these rules, the environments are minimized as follows. In the former rule, only the 'T-relevant' part of Γ', Δ is taken. In the latter, the extension of Γ, Γ' with Δ is omitted if α does not occur in $\vec{S}, T$ respectively.)

W.r.t. data symbols, the environment $\mathcal{E}$ contains labelled variants of the conventional types, supplied with propagation constraints. For example,

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

Like in the conventional case, $\mathcal{E}$ should be consistent with the rewrite rules. Here, $\mathcal{E}$ is said to be $\mathcal{R}$-OK if for each rule

$$\mathbf{F}(A_1, \ldots, A_k) \rightarrow E,$$

say with $\mathcal{E}$ containing $\mathbf{F} : \vec{S}|\Gamma \rightarrowtail T|\Delta$ there exist B, Γ' such that

$$B \mid \Gamma' \ \vdash^{=}_{\mathcal{E}} \ \vec{A} : \vec{S} \mid \Gamma, \qquad B \mid \Gamma' \ \vdash_{\mathcal{E}} \ E : T \mid \Delta,$$

where $\vdash^{=}_{\mathcal{E}}$ denotes derivability via (variable) and (application), without coercions. Note that if the rule contains no patterns ($\mathbf{F}\vec{x} \rightarrow E$) the above conditions simplify to

$$\vec{x} : \vec{S} \mid \Gamma \ \vdash_{\mathcal{E}} \ E : T \mid \Delta.$$

Also for uniqueness types, we prove that the subject reduction property holds. Moreover, there is a notion of principal uniqueness typing.

5. Concluding Remarks

We have presented a restriction of *Clean*'s uniqueness typing system in natural deduction style. Several refinements, such as higher-order functions and order-of-evaluation dependent reference analysis, will be formulated in this framework. 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 (1994). The present framework is likely to admit more direct proofs of these results. Finally, the relation with the approach of Benton (1994) and others will be investigated.

References

Ariola, Z.M. and J.W. Klop (1993). Equational term graph rewriting, Draft.

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). Uniqueness typing in theory and practice. To appear in: Proceedings PLILP'95.

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

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.

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.