Equality-Based Uniqueness Typing

Edsko de Vries*[1], Rinus Plasmeijer[2], and David M Abrahamson[1]

[1] Trinity College Dublin, Ireland, {devriese,david}@cs.tcd.ie
[2] Radboud Universiteit Nijmegen, Netherlands, rinus@cs.ru.nl

Abstract

Uniqueness typing can be used to add side effects to a functional programming language without losing referential transparency. Unfortunately uniqueness types often involve implications between uniqueness attributes, which complicates type inference and incorporating modern extensions such as arbitrary rank types. In this paper we show how to avoid these difficulties by recoding attribute inequalities as attribute equalities, and use this technique to define an arbitrary rank uniqueness type system.

1 INTRODUCTION

Referential transparency is an important feature of languages such as *Clean* and *Haskell* and is treasured because it facilitates reasoning about programs. A consequence of insisting on referential transparency is that functions must not be allowed to modify their arguments. For example, given the definition of *split* ($\triangle$):

$$f \triangle g = \lambda x \cdot (f\,x, g\,x)$$

we would expect to be able to prove that for all functions f,

$$\mathrm{snd} \circ (f \triangle \mathrm{id}) = \mathrm{id}$$

but this will only hold if f does not modify its argument. It *is* however safe for a function to modify its argument if the function has the sole reference to that argument. This is the basis of substructural type systems such as *Clean*'s uniqueness type system and the one we present here. As an example, consider a function *clearArray* that sets all values in an array to zero. Since *clearArray* will destructively modify its argument, it has the following type:

$$\mathrm{clearArray} :: \mathrm{Array}^{\bullet} \xrightarrow[\times]{u_f} \mathrm{Array}^{\bullet}$$

The details of this type will become clear in the rest of this paper. Suffice to say at this point that the uniqueness attribute ($\bullet$) in the domain of the function type indicates that *clearArray* requires a *unique* reference to an array; likewise, the codomain of the function indicates that *clearArray* promises to return a unique reference to an array. An expression such as *clearArray* $\triangle$ *id* then is ill-typed because *clearArray* must share its argument with *id*. We will consider the type of $\triangle$ in Section 3.3.

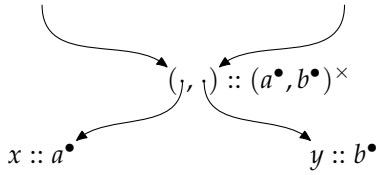
*Supported by the Irish Research Council for Science, Engineering and Technology.

Uniqueness typing is a more fundamental approach to adding side effects to a functional language than the monadic approach used in *Haskell*, since uniqueness typing can be used to *typecheck* the definition of the monad. Besides, using uniqueness typing we can write more informative types: a function in the `IO` monad could have any effect at all, whereas *clearArray* is very explicit about what it affects. Finally, functions that require their arguments to be unique can be applied and composed like any other function, but functions with a monadic type require quite a different style of programming. Of course, monads are useful even in a language with uniqueness typing, but are better regarded as a *design pattern* rather than "the way to do side effects".

However, uniqueness types often have constraints (implications between uniqueness attributes) associated with them. In *Clean*, for example, *fst* (the function which returns the first element of a pair) has type

$$\text{fst} :: (a^u, b^v)^w \to a^u, [w \leq u, w \leq v]$$

The constraint $[w \leq u]$ denotes that w must be unique if u is (u implies w). To understand the need for this constraint, suppose we have a pair with elements $x :: a$ and $y :: b$. The only references to these elements are from this pair, so a and b get a unique ($\bullet$) attribute. Further, suppose that there are two references to the pair, making the attribute of the type of the pair non-unique ($\times$). Visually:



If we could extract a unique element from a non-unique pair, we could extract x from the pair and modify it. But then the value of x as seen through the second reference will also change, and referential transparency is lost. So, we can only extract a unique element from a container if the container is unique itself $(w \leq u)$.

Although these constraints are important, they complicate the work of the type checker (unification cannot deal with inequalities) and make extending the type system to support modern features such as arbitrary rank types difficult (Section 6.3). In this paper we show how we can recode the attribute inequalities as attribute equalities. This results in a uniqueness type system which is sufficiently like the Hindley/Milner type system that standard type inference algorithms can be applied, and that modern extensions can easily be incorporated using existing techniques.

2 SHARING ANALYSIS

The typing rules we will present in this paper depend on a sharing analysis which marks variable uses as exclusive ($\odot$) or shared ($\otimes$). This sharing analysis could be more or less sophisticated [1], but if in any derivation of the program the same

variable could be evaluated twice, it must be marked as shared. In this paper, we assume sharing analysis has been done, leaving a formal definition to future work. Here we consider a few simple examples only. The identity function is marked as

$$\mathrm{id} = \lambda x \cdot x^{\odot}$$

since there is only one reference to x in the body of *id*. In the definition of *split*, however, there are two references to the same variable, which must therefore be marked as shared:

$$f \triangle g = \lambda x \cdot (f^{\odot} x^{\otimes}, g^{\odot} x^{\otimes})$$

The sharing analysis does not make a distinction between variables that correspond to functions and variables that correspond to function arguments. For example, the function *twice* is marked as

$$\mathrm{twice} = \lambda f \cdot \lambda x \cdot f^{\otimes} (f^{\otimes} x^{\odot})$$

3 TYPING THE CORE λ−CALCULUS

We present a uniqueness type system for the lambda calculus which does not involve implications. The expression and type language are defined in Figure 1. The expression language is the standard lambda calculus, except that variables are marked as exclusive ($x^{\odot}$) or shared ($x^{\otimes}$). The type language includes base types, type variables and the function space, all of which get a uniqueness attribute indicating whether there is more than one reference to a term. For instance, a term of type $\mathrm{Int}^{\bullet}$ is a unique integer (only one reference), but a term of type $\mathrm{Int}^{\times}$ is a non-unique integer (unknown number of references). We call such a type together with its uniqueness attribute an *attributed type*. The domain and codomain of the arrow (function space constructor) are both attributed types, and the arrow itself gets *two* attributes: the "normal" uniqueness attribute ν (indicating whether there is more than one reference to the function) and an additional "closure attribute" ν_c, which is the disjunction of all attributes on the types of the elements in the closure of the function. This is further discussed in Section 3.2.

We treat a uniqueness attribute as a boolean expression, reading True (unique) for "$\bullet$" and False (not unique) for "$\times$", and allow for arbitrary boolean expressions involving variables, negation, conjunction and disjunction as uniqueness attributes[1]. This definition of attributes is different from their definition in our previous paper [2] where (like in *Clean*) we only allow for unique, non-unique and variables. It may not be immediately obvious why this is useful, but as it turns out, all the improvements of the system as presented in this paper over the previous are made possible by this one change in the type language.

The typing relation itself takes the form

$$\Gamma \vdash e : \tau^{\nu}|_{fv}$$

[1] Although the typing rules only introduce disjunctions, unification may introduce more complicated types also involving conjunctions and negation.

e	::=	expression	ν	::=	attribute
	$x^{\odot}$	variable (exclusive)		u	attribute variable
	$x^{\otimes}$	variable (shared)		$\bullet$	unique
	$\lambda x \cdot e$	abstraction		$\times$	non-unique
	$e\,e$	application		$\neg\nu$	negation
				$\nu_1 \& \nu_2$	conjunction
τ^{ν}	::=	attributed type		$\nu_1 \mid \nu_2$	disjunction
	B^{ν}	base type			
	a^{ν}	type variable			
	$\tau_1^{\nu_1} \xrightarrow[\nu_c]{\nu} \tau_2^{\nu_2}$	function space			

FIGURE 1. Expression and type language for the core system

which reads as "in environment Γ, expression e has attributed type τ^{ν}; the attributes on the types of the free variables in e are *fv*". We represent *fv* as a relation $Var \times Attribute$; its purpose will become clear when we discuss the rule for abstraction in Section 3.2. The environment maps expression variables to attributed types.

Note the conspicuous absence of constraints in the type language. We will explain how we deal with this when we discuss the individual typing rules.

3.1 Variables

To check that a variable x marked as exclusive has attributed type τ^{ν}, we simply look up the variable in the environment[2]. For shared variables, we need to correct the type found in the environment to be non-unique. In both cases we also record the uniqueness attribute of the type of the variable (see Section 3.2).

$$\frac{}{\Gamma, x : \tau^{\nu} \vdash x^{\odot} : \tau^{\nu}|_{(x,\nu)}}\ \textsc{Var}^{\odot} \qquad \frac{}{\Gamma, x : \tau^{\nu} \vdash x^{\otimes} : \tau^{\times}|_{(x,\times)}}\ \textsc{Var}^{\otimes}$$

$\textsc{Var}^{\otimes}$ does not require the type *in the environment* to be non-unique. This effectively means that variables can lose their uniqueness[3]. For example, consider the function $mkPair = \lambda x \cdot (x^{\otimes}, x^{\otimes})$. Both components of the pair point to the same element, which is therefore non-unique by definition. Thus the type of *mkPair* is

$$\text{mkPair} :: a^{u} \xrightarrow[\times]{u_f} (a^{\times}, a^{\times})^{v}$$

The attributes on the arrow will be explained in the next section.

[2]When a variable usage is marked as exclusive, that does not automatically make its type unique; for example, the identity function $\lambda x \cdot x^{\odot}$ has type $a^{u} \to a^{u}$, not $a^{u} \to a^{\bullet}$. In other words, sharing analysis just notes that there is only one reference to x in the body of the identity function; however, when a non-unique argument is passed to *id*, it will still be non-unique when it is returned again.

[3]This is also the main difference between a uniqueness type system and an affine type system, where variables are either affine or not, but cannot lose their "affinity".

3.2 Abstraction

Before we discuss the rule for abstraction, we must first point out a subtlety due to partial application. Consider the function that returns the first of its two arguments:

$$\text{const} = \lambda x \cdot \lambda y \cdot x^{\odot}$$

Temporarily ignoring the attributes on arrows, *const* has type

$$\text{const} :: a^u \to b^v \to a^u$$

Given *const*, what would be the type of

$$\text{funnyMkPair} = \lambda x \cdot \text{let } f = \text{const } x^{\odot} \text{ in } (f^{\otimes} 1, f^{\otimes} 2)$$

It would seem that since f has type $b^v \to a^u$, this term has type

$$\text{funnyMkPair} :: a^u \to (a^u, a^u)^w$$

but this is clearly wrong: the elements in the result pair are shared, so the attribute on their types must be non-unique.

Recall from the introduction that if we want to extract a unique element from a container, the container must be unique itself. When we execute a function, the function can extract elements from its closure (the environment which binds the free variables in the function body). If any of those elements is unique, executing the function will involve extracting unique elements from a container (the closure), which must therefore be unique itself. Since we do not distinguish between a function and its closure in the lambda calculus, this means that the function must be unique. Thus a function needs to be unique on application (that is, a function can be applied only once) if the function can access unique elements from its closure.

The function type must therefore be modified to indicate whether there are any unique elements in the closure of the function. This is the purpose of the second uniqueness attribute on arrows (ν_c), which is the disjunction of all the attributes on the types of the elements in the closure of the function[4]. Going back to the example, the full type of f in the definition of *funnyMkPair* is therefore

$$f :: b^v \xrightarrow[u]{u_f} a^u$$

One way to read this type is that if you want a unique a to be returned from f, f must be unique on application. In the definition of *funnyMkPair*, f is not unique ($u_f = \times$) when applied since it is marked as shared, so the actual type of *funnyMkPair* is (see also Section 4):

$$\text{funnyMkPair} :: a^{\times} \xrightarrow[\times]{u_f} (a^{\times}, a^{\times})^w$$

[4]Those elements which are used. Unused elements can safely be ignored.

It should now be clear why the typing rules record the attributes on the free variables in an expression: we need this information to determine ν_c. Using $\bigvee fv$ to denote the disjunction of all attributes in the range of fv, and ${}^{\lhd}_{x}fv$ (domain subtraction) to denote fv with x removed from the domain of fv, we obtain

$$\frac{\Gamma, x : \tau_1^{\nu_1} \vdash e : \tau_2^{\nu_2}|_{fv}}{\Gamma \vdash \lambda x \cdot e : \tau_1^{\nu_1} \xrightarrow[\bigvee({}^{\lhd}_{x}fv)]{\nu_f} \tau_2^{\nu_2}|_{{}^{\lhd}_{x}fv}} \quad \textsc{Abs}$$

We must remove x from fv because x is not free in $\lambda x \cdot e$[5].

3.3 Application

As we have seen, some functions must be unique on application; this is enforced in the typing rule for application. Perhaps the most natural way to model this requirement is to use an inequality:

$$\frac{\Gamma \vdash e_1 : \tau_1^{\nu_1} \xrightarrow[\nu_c]{\nu_f} \tau_2^{\nu_2}|_{fv} \quad \Gamma \vdash e_2 : \tau_1^{\nu_1}|_{fv'} \quad \nu_f \leq \nu_c}{\Gamma \vdash e_1\, e_2 : \tau_2^{\nu_2}|_{fv \cup fv'}} \quad \textsc{ConstrApp}$$

The implication $[\nu_f \leq \nu_c]$ (ν_c implies ν_f) expresses the requirement that the function must be unique (ν_f) if it has any unique elements in its closure (ν_c). How can we model the requirement $\nu_f \leq \nu_c$ without using constraints? The easiest solution is to require that $\nu_f = \nu_c$:

$$\frac{\Gamma \vdash e_1 : \tau_1^{\nu_1} \xrightarrow[\nu_c]{\nu_c} \tau_2^{\nu_2}|_{fv} \quad \Gamma \vdash e_2 : \tau_1^{\nu_1}|_{fv'}}{\Gamma \vdash e_1\, e_2 : \tau_2^{\nu_2}|_{fv \cup fv'}} \quad \textsc{App}$$

While this rule is technically more restrictive than ConstrApp, in practice the programmer will not notice the difference. We will discuss this issue in more depth in Section 4.

3.4 Examples

We discuss two examples. First, we consider the type of $apply = \lambda f \cdot \lambda x \cdot f^{\odot} x^{\odot}$:

$$\text{apply} :: (a^u \xrightarrow[u_c]{u_c} b^v) \xrightarrow[\times]{u_f} a^u \xrightarrow[u_c]{u_{f'}} b^v$$

Unsurprisingly, *apply* takes a function f from a to b, and a term of type a, and returns a term of type b. Since *apply f* applies f, if f must be unique on application, it must be unique when passed as an argument to *apply* (in the type of *apply*, this requirement is encoded by specifying that f must have the same attribute below

[5] We use the Barendregt convention and assume that all bound variables are distinct.

and above the arrow). Finally, if f is unique, then *apply* f must be unique on application, since it can extract a unique element from its closure (to wit, f).

The type of $\triangle$, discussed in the introduction, may look a bit intimidating, but is only slightly more complicated:

$$\triangle :: (a^{\times} \xrightarrow[u_1]{u_1} b^{v}) \xrightarrow[\times]{uf} (a^{\times} \xrightarrow[u_2]{u_2} c^{w}) \xrightarrow[u_1]{uf'} a^{u} \xrightarrow[u_1|u_2]{uf''} (b^{v}, c^{w})^{z}$$

In words, $\triangle$ wants two functions f and g, which return a b^v and a c^w, given a non-unique a, and returns a pair of type $(b^v, c^w)^z$. If either f or g must be unique on application, then they must be unique when they are passed as arguments to $\triangle$, as $\triangle$ will apply them. Finally, $f \triangle g$ must itself be unique on application when either f or g is unique, because if they are, $f \triangle g$ will be able to extract unique elements from its closure (i.e., f and g) when it is applied. The function *clearArray* from the introduction cannot be passed as an argument to $\triangle$ since it does not accept non-unique arguments (Array$^{\bullet}$ does not unify with $a^{\times}$).

4 REFLECTION ON THE CORE SYSTEM

In Section 3.3, we removed the implication from CONSTRABS by replacing the implication $[\nu_f \leq \nu_c]$ by $[\nu_f = \nu_c]$. It is possible to remove the implication *without* giving a more restrictive rule by using disjunction with a free variable:

$$\frac{\Gamma \vdash e_1 : \tau_1^{\nu_1} \xrightarrow[\nu_c]{\nu_f|\nu_c} \tau_2^{\nu_2}|_{fv} \quad \Gamma \vdash e_2 : \tau_1^{\nu_1}|_{fv'}}{\Gamma \vdash e_1\, e_2 : \tau_2^{\nu_2}|_{fv \cup fv'}} \quad \text{APP}'$$

When $\nu_c = \times$, $\nu_f|\nu_c$ reduces to ν_f (a free variable), but when $\nu_c = \bullet$, $\nu_f|\nu_c$ reduces to $\bullet$. So, when $\nu_c = \times$ (when there are no unique elements in the function closure), the function may or may not be unique, but when $\nu_c = \bullet$, the function *must* be unique, which is exactly what the constraint $[\nu_f \leq \nu_c]$ specified. We nevertheless prefer rule APP (requiring that $\nu_f = \nu_c$), since it leads to more readable types. For example, based on rule APP$'$, *split* would have the type

$$\triangle :: (a^{\times} \xrightarrow[u_{a_1}]{u_{f_1}|u_{a_1}} b^{v}) \xrightarrow[\times]{u_f} (a^{\times} \xrightarrow[u_{a_2}]{u_{f_2}|u_{a_2}} c^{w}) \xrightarrow[u_{f_1}|u_{a_1}]{u_{f'}} a^{u} \xrightarrow[u_{f_1}|u_{a_1}|u_{f_2}|u_{a_2}]{u_{f''}} (b^{v}, c^{w})^{z}$$

However, we claimed that rule APP is not as restrictive as it may seem. An expression will be rejected by APP but allowed by APP$'$ if and only if the function that we are applying is unique, but does not have any unique elements in its closure; so, if we have an expression $f\,x$ where f has type

$$f :: a^{u} \xrightarrow[\times]{\bullet} b^{v}$$

Clearly $\bullet$ does not unify with $\times$, so rule APP will reject this application. The corresponding error message will be a bit mystifying: "*The function you are applying*

is too unique. Use it more often!". Of course, this is a consequence of replacing the implication by an equality. However, barring type annotations, that type will never be inferred for a term! Instead, the following type would be inferred:

$$f :: a^u \xrightarrow[\times]{u_f} b^v$$

That is, the function will be polymorphic in its uniqueness rather than actually be unique. None of the typing rules even mention $\bullet$ anywhere! The typing rules force terms to be non-unique if they are shared, but they never force them to be unique. Given the latter type of f, rule APP has no difficulty typing the application, since u_f trivially unifies with $\times$.

The reader might wonder why it is useful to distinguish between the uniqueness of the function and the (disjunction of) the uniqueness of the elements in the closure of the function, if we insist that they must be the same on application. It is in fact possible to collapse these two attributes into the one attribute, and use the uniqueness attribute of the function for both. Then a function must be unique if it has any unique elements in its closure, and must *remain* unique. This is the approach taken in *Clean*, but it complicates the type system. For example, the function *mkPair* from Section 3.1 must be assigned the type $a^\times \xrightarrow[\times]{u_f} (a^\times, a^\times)^v$ instead of $a^u \xrightarrow[\times]{u_f} (a^\times, a^\times)^v$, because the latter type would allow us to duplicate a function with unique elements in its closure (and then apply that function twice). To make this type less restrictive, *Clean* then introduces subtyping, but that brings complications of its own. By distinguishing between the two attributes (which really do represent different properties of the function), the requirement that a function with unique elements in its closure must be unique on application becomes a local requirement in the rule for application, and does not complicate the rest of the type system.

Finally, the reader may have been surprised and expected the type inferred for *funnyMkPair* to be the same as the type for *mkPair*. The reason that it is not (but is more restrictive) is due to rule VAR$^\odot$. When a variable usage is marked as exclusive, rule VAR$^\odot$ states that the type of the variable is equal to the type listed for the variable in the environment. It is possible to relax this rule to

$$\frac{}{\Gamma, x : \tau^{\nu|\nu'} \vdash x^\odot : \tau^\nu|_{(x,\nu)}} \quad \text{VAR}'^\odot$$

With this rule a variable whose type is listed as non-unique in the environment must be non-unique (as before), but when the type of the variable in the environment is listed as unique, this rule places no restrictions on the attribute of the type derived for x (if $\nu|\nu' = \bullet$, either ν or ν' must be unique, but they do not both have to be unique). With VAR$'^\odot$ *funnyMkPair* will indeed have the same type as *mkPair*, but at a cost: types become more complicated. For example, the identity function would have the type

$$\lambda x \cdot x^\odot :: a^{u|v} \xrightarrow[\times]{u_f} a^u$$

which is correct, but perhaps more difficult to understand than the type derived for the identity function using rule VAR$^{\odot}$ ($a^u \xrightarrow[\times]{u_f} a^u$).

5 TYPE INFERENCE

One advantage of removing constraints from the type language is that standard inference algorithms (such as algorithm $\mathcal{W}$ [3]) can be applied without any modifications. The inference algorithm will depend on a unification algorithm, which must be modified in two ways. It must treat a unification goal $\tau_1^{\nu_1} \doteq \tau_2^{\nu_2}$ as two separate goals $\tau_1 \doteq \tau_2$ and $\nu_1 \doteq \nu_2$ (in other words, base types and their attributes must be unified independently), and it must be adapted to deal with boolean expressions. The rest of this section explains how boolean unification works.

Suppose we have two terms g and h

$$g :: a^{\bullet} \xrightarrow[\times]{u_f} \ldots \qquad h :: a^{u|v}$$

Should the application $g\,h$ be allowed? If so, we must be able to unify $u|v$ and $\bullet$. Of course, this equation has many solutions, for example

$$\begin{bmatrix} u \mapsto \bullet \\ v \mapsto v \end{bmatrix} \qquad \begin{bmatrix} u \mapsto u \\ v \mapsto \bullet \end{bmatrix} \qquad \begin{bmatrix} u \mapsto \bullet \\ v \mapsto \bullet \end{bmatrix}$$

(Recall that we treat attributes as boolean expressions.) Unfortunately none of the solutions listed above is most general, and it not obvious that the equation $u|v = \bullet$ even has a most general unifier, which means we would lose principal types. Fortunately, unification in a boolean algebra is unitary [4]. In other words, if a boolean equation has a solution, it has a most general solution. In the example, one most general solution is

$$\begin{bmatrix} u \mapsto u \\ v \mapsto v|\neg u \end{bmatrix}$$

There are two well-known algorithms for unification in a boolean algebra, known as Löwenheim's formula and successive variable elimination. For the core system from Section 3 either will work, but when we introduce arbitrary rank types (Section 6), only successive variable elimination is practical[6]. The description of successive variable elimination we give here combines the methods from [4] and [5]. Switching temporarily to the usual notation for boolean algebra (using $+$ for

[6]Löwenheim's formula maps any unifier to a most general unifier, reducing the problem of finding an mgu to finding a specific unifier. For the two-element boolean algebra, that is very simple (just try all possible instantiations of the variables) but it is not so easy in the presence of skolem constants (Section 6.3). Skolem constants introduce new elements into the boolean algebra, making it much more difficult to guess ground unifiers. For example, assuming that u_R and v_R are skolem constants, and w is a uniqueness variable, the equation $u_R|v_R \doteq w$ has an obvious solution $[w \mapsto u_R|v_R]$, but we can no longer guess this solution by instantiating all variables to either true ($\bullet$) or false ($\times$).

$$
\begin{aligned}
&\texttt{unify}_0 :: \texttt{BooleanAlgebra}\ a \Rightarrow [Var] \to a \to (\texttt{Subst}\ a, a) \\
&\texttt{unify}_0\ []\ t = ([], t) \\
&\texttt{unify}_0\ (x : xs)\ t = (s_t \cup s_e, cc) \\
&\quad \texttt{where} \\
&\qquad s_t = [x \mapsto s_e\, t_0 + x \cdot s_e\, (\neg t_1)] \\
&\qquad (s_e, cc) = \texttt{unify}_0\ xs\ (t_0 \cdot t_1) \\
&\qquad t_0 = [x \mapsto 0]\ t \\
&\qquad t_1 = [x \mapsto 1]\ t
\end{aligned}
$$

FIGURE 2. Boolean unification

disjunction, $\cdot$ for conjunction, $\neg$ for negation, and 1 and 0 for True and False), to unify two terms p and q of a boolean algebra it suffices to unify

$$t = (p \cdot \neg q) + (\neg p \cdot q) = 0$$

This is implemented by $\texttt{unify}_0$, shown in Figure 2, which gets a term t in a boolean algebra a and the list of free variables in t as input, and returns a substitution and the "consistency condition", which will be zero if unification succeeded.

6 ARBITRARY RANK TYPES

We claim in this paper that our core uniqueness system is sufficiently similar to a standard Hindley/Milner type systems that modern extensions can be added without much difficulty. To substantiate this claim we show in this section how to extend the core type system to support arbitrary rank types using the techniques described in a recent paper by Peyton Jones *et al.* [6].

Section 6.1 explains what arbitrary rank types are and outlines how they are dealt with. This section is not new material, but serves as background material only. Section 6.2 explains how we must modify the typing rules from [6] to deal with uniqueness, and Section 6.3 explains why these modifications are much simpler in a system without inequalities than in a system with inequalities.

6.1 Arbitrary rank types

The core type system described in Section 3 does not have an explicit notion of universal quantification. When we say that the identity function has the type

$$\mathrm{id} = \lambda x \cdot x :: a^u \xrightarrow[\times]{u_f} a^u$$

what we mean is that for any instantiation of a, u and u_f, the identity function has that (instantiated) type. But this is a meta-level notion: the type language defined in Figure 1 does not allow universal quantification over type or uniqueness variables.

We can make universal quantification an object-level notion by introducing "type schemes": (attributed) types together with a list of universally quantified type (and uniqueness) variables.

$$\sigma ::= \forall \overline{a}, \overline{u}.\tau^{\nu} \quad \text{type scheme}$$

The typing rules can then be modified to assign a type scheme, rather than a type, to an expression. For example, the type scheme assigned to *id* would be

$$\lambda x \cdot x :: \forall a\, u\, u_f.a^u \xrightarrow[\times]{u_f} a^u$$

So far we have not gained much by introducing type schemes, but we can go one step further. We can modify the type language so that the domain of the function type constructor becomes a type scheme σ, rather than an (attributed) type τ^{ν}:

$\tau^{\nu} ::=$		attributed type
	B^{ν}	base type B
	a^{ν}	type variable
	$\sigma_1 \xrightarrow[\nu_c]{\nu} \tau_2^{\nu_2}$	function space

With this change we have suddenly gained a lot more expressive power in the type system. The example given in [6] (in the context of a type without support for uniqueness) is

$$g :: (\forall a.[a] \to [a]) \to ([\text{Bool}], [\text{Int}])$$
$$g = \lambda f \cdot (f\,[\text{True}, \text{False}], f\,[1,2,3])$$

In this example, g is a function that takes another function f as argument and applies f to two lists *of different types*. This example is type-correct because g insists that the type of f must be $([a] \to [a])$ *for all* type variables a, which is only possible because we allow for type schemes in the domain of the function type constructor.

Unfortunately type inference for higher rank types (types with nested universal quantifiers) is undecidable, but we can support higher rank types by combining type inference with type checking. We therefore use a slightly modified typing relation

$$\Gamma \vdash_{\delta} e : \tau^{\nu}|_{fv}$$

This is different from the typing relation in Section 3 in two ways: the environment Γ now maps expression variables to type schemes (not types), and we introduce a "typing mode" δ: $\Uparrow$ for type inference or $\Downarrow$ for type checking.

$$\frac{\vdash_\delta^{\text{inst}\sigma} \sigma \le \tau^\nu}{\Gamma, x:\sigma \vdash_\delta x^\odot : \tau^\nu|_{(x,\nu)}}\ \textsc{Var}^\odot \qquad \frac{\vdash_\delta^{\text{inst}\sigma} \sigma \le \tau^\nu}{\Gamma, x:\sigma \vdash_\delta x^\otimes : \tau^\times|_{(x,\times)}}\ \textsc{Var}^\otimes$$

$$\frac{\Gamma, x:\forall.\tau_1^{\nu_1} \vdash_\Uparrow t : \tau_2^{\nu_2}|_{fv}}{\Gamma \vdash_\Uparrow \lambda x\cdot t : \forall.\tau_1^{\nu_1} \xrightarrow[\vee^{\lhd_x fv}]{u_f} \tau_2^{\nu_2}|_{\lhd_x fv}}\ \textsc{Abs}^\tau \qquad \frac{\Gamma, x:\sigma \vdash_\Downarrow t : \tau^\nu|_{fv}}{\Gamma \vdash_\Downarrow \lambda x\cdot t : \sigma \xrightarrow[\vee^{\lhd_x fv}]{u_f} \tau^\nu|_{\lhd_x fv}}\ \textsc{Abs}^\sigma$$

$$\frac{\Gamma \vdash_\Uparrow t : \sigma \xrightarrow[\nu_c]{\nu_c} \tau'^{\nu'}|_{fv} \quad \Gamma \vdash_\Downarrow^{\text{gen}} u : \sigma|_{fv'} \quad \vdash_\delta^{\text{inst}\tau} \tau'^{\nu'} \le \tau^\nu}{\Gamma \vdash_\delta t\, u : \tau^\nu|_{fv \cup fv'}}\ \textsc{App}$$

$$\frac{\Gamma \vdash_\Downarrow^{\text{gen}} t : \sigma|_{fv} \quad \vdash_\delta^{\text{inst}\sigma} \sigma \le \tau^\nu}{\Gamma \vdash_\delta (t :: \sigma) : \tau^\nu|_{fv}}\ \textsc{Ann} \qquad \frac{\Gamma \vdash_\Uparrow^{\text{gen}} u : \sigma|_{fv} \quad \Gamma, x:\sigma \vdash_\delta t : \tau^\nu|_{fv'}}{\Gamma \vdash_\delta \text{let } x = u \text{ in } t : \tau^\nu|_{fv \cup (\lhd_x fv')}}\ \textsc{Let}$$

$$\frac{(\overline{a},\overline{u}) = ftuv(\tau^\nu) - ftuv(\Gamma) \quad \Gamma \vdash_\delta t : \tau^\nu|_{fv}}{\Gamma \vdash_\delta^{\text{gen}} t : \forall \overline{a},\overline{u}.\tau^\nu|_{fv}}\ \textsc{Gen} \qquad \frac{\vdash_\delta^{\text{inst}\rho} [\overline{a \mapsto \tau}][\overline{u \mapsto \nu}]\tau^\nu \le \tau'^{\nu'}}{\vdash_\delta^{\text{inst}\sigma} \forall \overline{a},\overline{u}.\tau^\nu \le \tau'^{\nu'}}\ \textsc{Inst}^\sigma$$

$$\frac{}{\vdash_\Uparrow^{\text{inst}\rho} \tau^\nu \le \tau^\nu}\ \textsc{Inst}^\Uparrow \qquad \frac{\vdash^{\text{subs}} \tau^\nu \le \tau'^{\nu'}}{\vdash_\Downarrow^{\text{inst}\rho} \tau^\nu \le \tau'^{\nu'}}\ \textsc{Inst}^\Downarrow$$

$$\frac{(\overline{a},\overline{u}) \notin ftuv(\sigma) \quad \vdash^{\text{subs}} \sigma \le \tau^\nu}{\vdash^{\text{subs}} \sigma \le \forall \overline{a},\overline{u}.\tau^\nu}\ \textsc{Skol} \qquad \frac{\vdash^{\text{subs}} [\overline{a \mapsto \tau}][\overline{u \mapsto \nu}]\tau_1^{\nu_1} \le \tau_2^{\nu_2}}{\vdash^{\text{subs}} \forall \overline{a},\overline{u}.\tau_1^{\nu_1} \le \rho_2^{\nu_2}}\ \textsc{Spec}$$

$$\frac{\vdash^{\text{subs}} \sigma_2 \le \sigma_1 \quad \vdash^{\text{subs}} \tau_1^{\nu_1} \le \tau_2^{\nu_2}}{\vdash^{\text{subs}} (\sigma_1 \xrightarrow[\nu_c]{\nu_f} \tau_1^{\nu_1}) \le (\sigma_2 \xrightarrow[\nu_c]{\nu_f} \tau_2^{\nu_2})}\ \textsc{Fun} \qquad \frac{}{\vdash^{\text{subs}} \tau^\nu \le \tau^\nu}\ \textsc{Mono}$$

FIGURE 3. Arbitrary Rank Typing Rules

The full typing rules are shown in Figure 3. A detailed discussion of the arbitrary rank typing rules is beyond the scope of this paper (in fact, a detailed discussion took 82 pages [6]), but we want to emphasize that the typing rules shown in this paper (which support uniqueness) closely resemble the original typing rules (which do not). We highlight the differences in the next section.

6.2 Modifications to deal with uniqueness

In this section we briefly highlight how a type system based on [6] must be modified to deal with uniqueness. Our starting point is the "bidirectional" typing rules from [6, Fig. 8], except for simplicity of presentation we do not show the rules for annotated lambda abstractions (rules AABS1 and AABS2), and we require types to be in *prenex form*: we allow for type schemes in the domain of the function type (as explained in the previous section), but not in the codomain. This choice is discussed in [6, Section 4.6.2], and simplifies the type system (in particular, the definition of skolemisation). We make the following modifications:

1. We refer to an attributed type τ^v wherever the original rules refer to a type τ.
2. We add a rule VAR$^{\otimes}$ to deal with sharing.
3. We record *fv*, the attributes on the free variables in a term, and remember to remove a variable from *fv* at all binding sites (rules ABS$^{\tau}$, ABS$^{\sigma}$ and LET).
4. Any rule that mentions the function space constructor ($\xrightarrow[v_c]{v_f}$) is modified to deal with v_f and v_c. The modifications to rules ABS$^{\tau}$, ABS$^{\sigma}$ and APP follow directly from the core system. Rule FUN compares two attributed types to check if one is at least as polymorphic as the other. For attributed types this is generally only true if both are equal (rule MONO), except rule FUN deals with the type schemes in the domain of the function type. However, the atributes on the arrow are simply part of the attributed types, and must therefore be equal (like in rule MONO).
5. Finally, all the rules that deal with type schemes are modified to allow for universal quantification of attribute variables in addition to universal quantification over type variables (rules GEN, INST$^{\sigma}$, SKOL and SPEC). The function *ftuv* returns the free type and uniqueness variables in its argument.

We argue that all of these modifications follow in a straightforward way given the core system we presented in Section 3, and moreover do not change the type system presented in [6] in any essential way: the structure of the type system (with the exception of rule VAR$^{\otimes}$) is still the exact same. Moreover, the implementation of the type system, including techniques such as skolemisation as described in [6, Sections 5 and 6] can be applied without any major modifications.

The typing rules do not include a rule for recursive let expressions. It is possible to add such a rule, but the current presentation of the rules makes it a bit awkward to express. It is not difficult to reorganize the rules to solve that problem, but that would make a superficial comparison between the type system presented in this paper and the original type system in [6] more difficult, so we opted not to. Either way, the rule for recursive let expressions must make sure that a term which is defined recursively gets a non-unique type (as it refers to itself [1]).

6.3 Complications due to inequalities

We have shown in the previous section that it is straight-forward to extend our core system with support for arbitrary rank types. This extension is not so trivial when the type system involves inequalities (constraints). In this section we explain why, and compare the type system in this paper with our previous type system, which did make use of inequalities [2].

In *Clean* constraints are never explicitly associated with types in the typing rules. Rather, the typing rules simply list the constraints as additional premises. However, that approach does not scale up to arbitrary rank types. When we generalize a type $\tau_a^{v_a}$ to a type scheme σ, $\tau_a^{v_a}$ may be constrained by a set of constraints $\mathcal{C}$. Those constraints should be associated with the type scheme σ, because if at a later stage we instantiate σ to get a type $\tau_b^{v_b}$, the same set of constraints should

apply to $\tau_b^{\nu_b}$ as well. Thus in [2] we defined a type scheme σ as

$$\forall \overline{x}.\tau^{\nu}, C$$

In other words, a type scheme is an attributed type τ^{ν}, together with a set of universally quantified (type and uniqueness) variables $\overline{x}$, and a set of constraints C. The typing rules then are careful to manipulate constraint sets. For example, the rule for instantiating a type scheme read

$$\frac{}{\forall \overline{x}.\tau^{\nu}, C \leq \mathcal{S}_x\tau^{\nu} | \mathcal{S}_x C} \quad \textsc{OldInst}$$

With this rule we can instantiate a type scheme to a type using a substitution $\mathcal{S}_x$, but only if the constraints associated with the type scheme are satisfied.

If we want to allow for arbitrary rank types we must modify the domain of the arrow (the function type constructor) to be a type scheme. Unfortunately that means that we now have constraints appearing in multiple places in type schemes. For example, we might have

$$\mathrm{id}' :: \forall a u u_f.(\forall.a^u, \emptyset) \xrightarrow[\times]{u_f} a^u, \emptyset = \lambda x \cdot x$$

We could add some syntactic sugar to make this type more readable (to get $a^u \xrightarrow[\times]{u_f} a^u$ or even $a^u \to a^u$), but that hides a more fundamental problem: the type of id' only accepts arguments of type a^u, if those arguments have type a^u *under the empty set of constraints*. If a term has type a^u only if a particular set of constraints is satisfied, that term cannot be used as an argument to id'. To get around this problem we need to introduce types that are polymorphic in their constraint sets. This is what we did in the previous paper. The type of *id* would then be

$$\mathrm{id} :: \forall a u u_f c.(\forall.a^u, c) \xrightarrow[\times]{u_f} a^u, c$$

which says that *id* accepts terms that have type a^u under the set of constraints c; the result then also has type a^u, if the same set of constraints is satisfied. This becomes particularly cumbersome for functions with many arguments, and especially for higher order functions (functions taking functions as arguments).

The definition of subsumption (checking whether one type scheme is at least as general as another) is also complicated by the presence of the constraint sets and constraint variables associated with type schemes. To check whether a type scheme σ_1 subsumes σ_2, we need to check whether the constraints associated with σ_2 logically entail σ_1. For details we refer to [2]; here we consider an example only. Suppose we have two functions f, g with types

$$f :: (\forall u v.a^u \xrightarrow[u_c]{u_f} b^v, \emptyset) \to \ldots$$
$$g :: a^u \xrightarrow[u_c]{u_f} b^v, [u \leq v]$$

Should the application $f\,g$ type-check? Intuitively, f expects to be able to use the function it is passed to obtain a b with uniqueness v (say, a unique b), independent of the uniqueness of a. However, g only promises to return a unique b if a is also unique; the application $f\,g$ should therefore be disallowed. Conversely, if we instead define f' and g' as

$$f' :: (\forall\, u\, v.a^u \xrightarrow[u_c]{u_f} b^v, [u \leq v]) \to \ldots$$
$$g' :: a^u \xrightarrow[u_c]{u_f} b^v, \emptyset$$

the application $f'\,g'$ *should* be allowed because the type of g' is more general than the type expected by f'. But it is not completely clear how to define subsumption in a completely general fashion. For example, suppose f was defined as

$$f :: (\forall\, u\, v.a^u \xrightarrow[u_c]{u_f} b^v, c_1 \cup c_2) \to \ldots$$

(Recall that c_1 and c_2 are constraint sets.) Then should the application $f\,g$ be allowed? Intuitively it should, since we can instantiate c_1 to $u \leq v$ and c_2 to the empty constraint (the constraint that is vacuously satisfied), but it is not easy to define this formally. When constraints are remodelled as boolean expressions, however, this problem is taken care of by boolean unification.

The fact that we do not have to do anything special to define subsumption in this paper is interesting, and further evidence for our claim that the core system is sufficiently similar to the Hindley/Milner type system that modern extensions can easily be incorporated. It is instructive to reconsider the last two examples. Recast in the new type system, the types of f and g are

$$f :: (\forall\, u\, v.a^u \xrightarrow[u_c]{u_f} b^v) \to \ldots$$
$$g :: a^{u|v} \xrightarrow[u_c]{u_f} b^v$$

where we have remodelled the implication $u \leq v$ as a disjunction $u|v$. Of course, by the same argument as the one used above, the application $f\,g$ should still be disallowed. This will be detected by the subsumption check. Part of the subsumption check will try to solve $u_R \doteq u|v$ and $v_R \doteq v$ (where u_R and v_R are skolem constants, that is, fixed but unknown attributes). Taken individually, each equation can be solved. However, as soon as we solve one, the other becomes insoluble and the subsumption check fails with an error message such as

`Cannot unify` v_R `and` $v\&u_R$

On the other hand, given the types of f' and g'

$$f' :: (\forall\, u\, v.a^{u|v} \xrightarrow[u_c]{u_f} b^v) \to \ldots$$
$$g' :: a^u \xrightarrow[u_c]{u_f} b^v$$

subsumption will need to solve the equations $u_R|v_R \doteq u$ and $v_R = v$, which have a trivial solution $[u \mapsto u_R|v_R, v \mapsto v_R]$, and the application $f'\,g'$ is therefore accepted. So, where we needed to check for logical entailment before, the technique of skolemisation (which we needed anyway) will suffice in the new system.

7 CONCLUSIONS

Uniqueness typing can be used to add side effects such as destructive updates to a pure functional language without loosing referential transparency. Uniqueness types in the type system of *Clean* or in the system we proposed in a previous paper [2] often involve inequalities (implications) between uniqueness attributes. This complicates type inference and makes incorporating modern extensions such as arbitrary rank types difficult; *Clean* for example does not fully support arbitrary rank types. We have shown how to avoid these difficulties by recoding attribute inequalities as attribute equalities. The new type system is sufficiently similar to the standard Hindley/Milner type system that standard inference algorithms can be applied, and modern extensions such as arbitrary rank types or GADTs can be incorporated using existing techniques [7], although the length of this paper did not permit us to demonstrate the latter.

Future work includes a formalization of the type system, and an investigation into which syntactic conventions we can introduce to make the type system easier to use. We would like the programmer to be able to write type annotations, and the compiler to report inferred types, omitting uniqueness attributes unless considered relevant.

REFERENCES

[1] Barendsen, E., Smetsers, S.: Conventional and uniqueness typing in graph rewrite systems. Technical Report CSI-R9328, University of Nijmegen (1993)

[2] De Vries, E., Plasmeijer, R., Abrahamson, D.: Uniqueness typing redefined. In Horváth, Z., Zsók, V., Butterfield, A., eds.: Revised selected papers from IFL 2006, LNCS 4449. (2007)

[3] Damas, L., Milner, R.: Principal type-schemes for functional programs. In: POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM Press (1982) 207–212

[4] Baader, F., Niphow, T.: Term Rewriting and All That. Cambridge University Press (1998)

[5] Brown, F.M.: Boolean Reasoning, The Logic of Boolean Equations. Dover Publications, Inc. (2003)

[6] Peyton Jones, S., Vytiniotis, D., Weirich, S., Shields, M.: Practical type inference for arbitrary-rank types. Journal of Functional Programming **17**(1) (2007) 1–82

[7] Peyton Jones, S., Vytiniotis, D., Weirich, S., Washburn, G.: Simple unification-based type inference for GADTs. In: Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming. (2006) 50–61