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

We define a uniqueness type system for the core lambda calculus which, unlike *Clean*'s uniqueness system and the system we proposed in a previous paper [1], does not involve inequalities. We claim that this makes the type system sufficiently similar to the Hindley/Milner type system that standard type inference algorithms can be applied, and that it can easily be modified to incorporate modern extensions such as arbitrary rank types and generalized algebraic data types. We substantiate this claim by sketching out how such a system would be defined.

1 INTRODUCTION

Referential transparency (the principle that we can substitute equals for equals) is an important feature of pure functional programming languages such as *Clean* and *Haskell* and is treasured because it facilitates reasoning about programs. A direct 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

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

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 (given a primitive type *Array*). 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 bullet ($\bullet$) in the domain of the function type indicates that *clearArray* requires a *unique* reference to an array; likewise, the bullet in the codomain

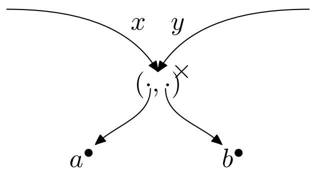
[⋆]Supported by the Irish Research Council for Science, Engineering and Technology: funded by the National Development Plan

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 and will be rejected by the type checker (we will consider the type of $\triangle$ in Section 3.3).

Uniqueness types in Clean (and in the type system we proposed previously [1]) often have constraints associated with them. In Clean, for example, the function *fst* that returns the first element of a pair has type

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

The constraint $[w \leq u]$ denotes that if u is unique, then w must be unique (u implies w)[1]. To understand the need for this constraint, suppose we have a pair with two elements of type a and 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 x and y to the pair. Since there is more than one reference to the pair, the attribute of the type of the pair is non-unique ($\times$). We can visualize this as follows:



In this example, if we were allowed to extract a unique element from a non-unique pair, we could extract a from the pair and pass it to a function f that expects a unique argument and modifies it. But then the value of a as seen through the other reference (y) will also change, and referential transparency is lost. Therefore, we can only extract a unique element from a container if the container is unique itself.

Although these constraints are evidently important, they complicate the work of the type checker (the heart of the typechecker is a unification algorithm, and unification cannot deal with inequalities) and make extending the type system to support modern features such as arbitrary rank types difficult (Section 6.4).

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 which can be extended easily using existing techniques to support for example arbitrary rank types and generalized algebraic data types (we outline such a system in Section 6).

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

[1]Perhaps the choice of the symbol $\leq$ is unfortunate. In logic $a \leq b$ denotes a implies b, whereas here $u \leq v$ denotes v implies u. Usage here conforms to *Clean* conventions.

be more or less sophisticated [2], 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:

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

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 $\lambda-$CALCULUS

In this section, we consider a uniqueness type system for the core lambda calculus, which is completely equality based (we do not use inequalities anywhere). The main typing relation used in this section is

$$\Gamma, u_{\gamma} \vdash e : \tau^{\nu}$$

which reads as "in environment Γ, given uniqueness attribute u_{γ}, expression e has attributed type τ^{ν}". The purpose of u_{γ} will become clear when we discuss the rule for abstraction in Section 3.2. The environment maps expression variables to attributed types.

The expression and type language for the core system are defined in Table 1. The expression syntax is the standard core lambda calculus, except that variables are marked as exclusive or shared. The type language includes base types, type variables and functions. The domain and codomain of the arrow (function constructor) are both attributed types, and the arrow itself gets *two* attributes: the "normal" uniqueness attribute ν (which indicates whether there is more than one reference to the function) and an additional attribute ν_a, discussed in Section 3.2.

The definition of attributes is different from their definition in our previous paper [1] (and from their definition in *Clean*, too). We treat a uniqueness attribute as a boolean attribute, reading True (unique: there is only one reference to the term) for $\bullet$ and False (not unique: unknown number of references) for $\times$. Then, we allow for arbitrary boolean expressions involving variables, negation, conjunction and disjunction as uniqueness attributes[2]. It may not be immediately obvious why

[2]Uniqueness attributes with free variables only have a meaning in an environment where those free variables are bound.

e ::=	expression	ν ::=	attribute
$x^{\odot}$	variable (exclusive)	u	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} ::=	type	$\nu_1 \mid \nu_2$	disjunction
B^{ν}	base type B		
t^{ν}	variable		
$\tau_1^{\nu_1} \xrightarrow[\nu_a]{\nu} \tau_2^{\nu_2}$	function		

TABLE 1. Expression and type language for the core system

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.

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[3]. For shared variables, we need to correct the type found in the environment to be non-unique:

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

$\textsc{Var}^{\otimes}$ does not require the type *in the environment* to be non-unique. This effectively means that variables can lose their uniqueness[4]. For example, consider the function *mkPair* $= \lambda x \cdot (x^{\otimes}, x^{\otimes})$. The body of *mkPair* can be visualized as

$$\begin{tikzcd} (\cdot,\cdot)^{v} \arrow[d, bend right] \arrow[d, bend left] \\ a^{\times} \end{tikzcd}$$

In other words, both components of the pair point to the same element, which is therefore non-unique by definition. Thus, the type of *mkPair* is

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

[3]When a variable is marked as exclusive, that does not automatically make it unique; for example, the identity function $\lambda x \cdot x^{\odot}$ has type $a^{u} \to a^{u}$, not $a^{u} \to a^{\bullet}$.

[4]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"

(The attributes on the arrow will be explained in Section 3.2). The previous version of our type system [1] would assign the same type to this term, but in *Clean* it would be assigned a different type as explained in [1] (briefly, type variables can be function valued, and functions are not allowed to lose their uniqueness in Clean; therefore, type variables cannot lose their uniqueness either).

3.2 Abstraction

Before we discuss the rule for abstraction, we must first point out a subtlety due to currying. 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

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

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 $b^v \longrightarrow a^u$, this term has type

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

but this is clearly wrong: the a's in the result type are shared within the pair, so they must be non-unique. How is this problem solved? In addition to their normal uniqueness attribute, functions have an additional attribute ν_a. The purpose of this attribute is to indicate whether the function needs to be unique on application: a function with type $a^u \underset{\times}{\longrightarrow} b^v$ does not need to be unique when it is applied, but a function with type $a^u \underset{\bullet}{\longrightarrow} b^v$ does.

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 body of the function). 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. To summarize, 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.

Going back to the example, the full type of f in the definition of *funnyMkPair* is therefore

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

In other words, if you want a unique a to be returned from f, f must be unique when it is applied. In the definition of *funnyMkPair*, f is not unique when applied since it is marked as shared ($u_f = \times$), so as expected *funnyMkPair* has the same type as *mkPair*:

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

It should be clear from the preceding discussion that to be able to type a function, we need to know whether the function will be able to extract unique elements from its closure. This is indicated by u_γ in the typing relation, and will be the case if the function is defined within another function, where the argument to the outer function is unique. Initially, u_γ will be $\times$, but as soon as we start typechecking the body of a lambda abstraction, $u_{\gamma'}$ must be unique when the argument to the lambda abstraction is unique. Using constraints, we can therefore give the following rule for lambda abstraction:

$$\frac{(\Gamma, x : \tau_1^{\nu_1}), u_{\gamma'} \vdash e : \tau_2^{\nu_2} \qquad \nu_a \leq u_\gamma, u_{\gamma'} \leq \nu_1, u_{\gamma'} \leq u_\gamma}{\Gamma, u_\gamma \vdash \lambda x \cdot e : \tau_1^{\nu_1} \xrightarrow[\nu_a]{\nu_f} \tau_2^{\nu_2}} \quad \textsc{ConstrAbs}$$

The constraints specify that ν_a must be unique when u_γ is, and $u_{\gamma'}$ (used to type the body of the abstraction) must be unique when either u_γ is or ν_1 is. Without making the system more restrictive, we can remove these constraints by stating that ν_a must be *equal* to u_γ, and by using the disjunction of u_γ and ν_1 to type the body of the abstraction, thus:

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

3.3 Application

The rule for application must enforce that functions which must be unique when applied, are unique when applied. Again, we can use a constraint to express this property:

$$\frac{\Gamma, u_\gamma \vdash e_1 : \tau_1^{\nu_1} \xrightarrow[\nu_a]{\nu_f} \tau_2^{\nu_2} \qquad \Gamma, u_\gamma \vdash e_2 : \tau_1^{\nu_1} \qquad \nu_f \leq \nu_a}{\Gamma, u_\gamma \vdash e_1\, e_2 : \tau_2^{\nu_2}} \quad \textsc{ConstrApp}$$

How can we model the requirement $\nu_f \leq \nu_a$ without using constraints? The easiest solution is to require that $\nu_f = \nu_a$:

$$\frac{\Gamma, u_\gamma \vdash e_1 : \tau_1^{\nu_1} \xrightarrow[\nu_a]{\nu_a} \tau_2^{\nu_2} \quad \Gamma, u_\gamma \vdash e_2 : \tau_1^{\nu_1}}{\Gamma, u_\gamma \vdash e_1\, e_2 : \tau_2^{\nu_2}} \quad \textsc{App}$$

While this makes the type system more restrictive, that will not be very noticeable in practice; besides, it is possible to give a rule for application which is not more restrictive (and yet does not use constraints). Both these issues are discussed in Section 4.

3.4 Examples

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

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

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 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 (that is, f).

The type of $\triangle$, discussed in the introduction, 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 claimed that it is possible to give a rule for application which does not use constraints but is not more restrictive than OLDAPP. One solution is to use a disjunction with a free variable:

$$\frac{\Gamma, u_\gamma \vdash e_1 : \tau_1^{\nu_1} \xrightarrow[\nu_a]{\nu_f|\nu_a} \tau_2^{\nu_2} \qquad \Gamma, u_\gamma \vdash e_2 : \tau_1^{\nu_1}}{\Gamma, u_\gamma \vdash e_1\, e_2 : \tau_2^{\nu_2}}\ \textsc{App}'$$

When $\nu_a = \times$, $\nu_f|\nu_a$ reduces to ν_f (a free variable), but when $\nu_a = \bullet$, $\nu_f|\nu_a$ reduces to $\bullet$. So, when $\nu_a = \times$, the function may or may not be unique, but when $\nu_a = \bullet$, the function *must* be unique, which is exactly what the constraint $[\nu_f \leq \nu_a]$ specified. We nevertheless prefer rule APP (requiring that $\nu_f = \nu_a$), since it leads to simpler types. For example, based on rule APP′, *split* would have the type

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

which is still much better than the type our previous type system would assign, but rather complex all the same.

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 need to be unique on application; 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; something like: "*The function you are applying is too unique (please use it more than once)*". Of course, this is a direct consequence of replacing the implication by an equality. However, it is very unlikely that f has the type shown above! It is much more likely to have the type

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

That is, it is much more likely to be polymorphic in its uniqueness 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 $\times$.

In the presence of algebraic data types and pattern matching, we can apply a similar technique. Recall the type of *fst* in Clean (discussed in the introduction).

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

We can recode the constraints as follows:

$$\text{fst} :: (a^u, b^v)^{u|v|w} \xrightarrow[\times]{u_f} a^u$$

That is, the pair must be unique if either u or v is unique, but if neither u nor v is unique, the pair may or may not be unique (w). This is a faithful translation of the implication; we can also give a slightly more restrictive type:

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

This type requires the pair to be non-unique if both u and v are non-unique. However, as for functions, the pair is much more likely to be polymorphic in its uniqueness than actually be unique, so the fact that this (simpler) type of *fst* is more restrictive than it needs to be will not be very noticeable in practice. Finally, since *fst* does not refer to the second component of the pair at all, we could even give *fst* the type

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

which is arguably the most intuitive type we could give *fst*; however, this final type would require slightly more advanced typing rules to analyze which variables are referenced and which are not.

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.

Consider the function that returns the first of its three arguments.

$$\text{fst3} = \lambda x \cdot \lambda y \cdot \lambda z \cdot x^{\odot} :: a^u \xrightarrow[\times]{u_f} b^v \xrightarrow[u]{u_{f'}} c^w \xrightarrow[u|v]{u_{f''}} a^u$$

Given two variables m and n, with types a^u and b^v, we have

$$g = (\lambda x \cdot \lambda y \cdot \lambda z \cdot x^{\odot})\, m^{\odot}\, n^{\odot} :: c^w \xrightarrow[u|v]{u_{f''}} a^u$$

Now suppose we have a function h with the following type:

$$h :: (c^w \xrightarrow[\bullet]{u_{f''}} a^u) \to \ldots$$

Is the application hg well-typed? If it is, we must be able to unify $u|v$ and $\bullet$. Of course, there are various solutions to this equation, 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.) Since there are solutions, hg is certainly well-typed; but what *is* its type? None of the solutions listed above is most general, and it not so obvious that the equation $u|v = \bullet$ even has a most general unifier, which would mean that we lose principal types. Fortunately, it turns out that 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

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

Boolean unification has an even stronger property: if a boolean equation has a solution, it will have a *reproductive* unifier. Recall that for a unifier θ to be a most general unifier, we must have the property that for all other unifiers ζ,

$$\zeta = \zeta' \circ \sigma \qquad \text{for some unifier } \zeta'$$

A unifier σ is a reproductive unifier if for all other unifiers ζ,

$$\zeta = \zeta \circ \sigma$$

There are two well-known algorithms for unification in a boolean algebra, known as Löwenheim's formula and successive variable elimination. For the core type system defined in this paper, either technique will work, but when we scale the type system to arbitrary rank types (Section 6), only successive variable elimination is practical[5]. The description of successive variable elimination we give here is a combination of the methods described in [4] and [5], but is better suited than either for our purposes. Switching temporarily to the more usual notation for boolean expressions (as it makes the definitions clearer), to solve an equation $\nu_1 \doteq \nu_2$, it suffices to solve

$$(\nu_1 \cdot \nu_2') + (\nu_1' \cdot \nu_2) = 0$$

Successive variable elimination is then defined as follows. Let

$$t(x) = x' \cdot t(0) + x \cdot t(1)$$

and define $e = t(0) \cdot t(1)$. Then,

1. Every unifier of $t \approx 0$ is a unifier of $e \approx 0$.
2. If σ_e is a reproductive unifier of $e \approx 0$ and $x \notin \text{dom}(\sigma_e)$, then

   $$\sigma_t := \sigma_e \cup \{x \mapsto \sigma_e(t(0)) + x \cdot \sigma_e(t(1))'\}$$

   is a reproductive unifier of $t(x) \approx 0$.

6 ADVANCED FEATURES

The main claim of this paper is 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 have defined and implemented a much more advanced system based on the core system from Section 3, that supports arbitrary rank types and generalized algebraic data types, using techniques from two recent papers by Simon Peyton Jones *et al.* [6, 7].

Due to the limited scope and length of this paper, we cannot give the full details of the type system here but can only sketch out how a type system based on [6] and [7], but supporting only "normal" types, must be adapted to deal with uniqueness.

[5]Löwenheim's formula maps a ground unifier to an 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 6.4). Skolem constants introduce new elements into the underlying 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 an "ordinary" uniqueness attribute, 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 ($\times$).

We explain what arbitrary rank types and generalized algebraic data types are and (at a very high level) how they are dealt with in sections 6.1 and 6.2. We highlight the changes that need to be made to the typing rules to deal with uniqueness in section 6.3. Finally, we explain in Section 6.4 why these extensions are much simpler in a system without inequalities than in a system with inequalities.

6.1 Arbitrary rank types

The rank of a type is the depth at which universal quantifiers appear in the domain of functions. In most types, universal quantifiers appear only at the outermost level, for example

$$\mathrm{id} :: \forall a.a \rightarrow a$$

which is a type of rank 1. In higher rank types, we have nested universal quantifiers. For example [6],

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

In this example, g requires a function f that works on lists of type $[a]$ for all a (the rank of the type of g is 2). It is actually not that difficult to support arbitrary rank types, but the problem is that type *inference* is undecidable for types with rank $n > 2$. To solve that problem, the type checker must combine type inference with type checking, and higher rank types are only allowed if an explicit type signature is provided (like we did for the type of g).

6.2 Generalized algebraic data types

Generalized algebraic data types are a generalization of algebraic data types, where the programmer explicitly specifies the type of each constructor. As a simple example, here is a definition of a GADT that holds either a boolean or an integer:

```
data T :: * → * where
  TInt :: Int → T Int
  TBool :: Bool → T Bool
```

Since we can specify the types of the constructors manually, we can vary the argument of T for each constructor. This allows us to write the following function:

```
projT ::: T a → a
projT (TInt i) = i
projT (TBool b) = b
```

Without GADTs, we cannot not write this function because it could not be typed. This is only a simple example of GADTs; there are many more examples in the literature; see [7] for a number of references.

Apart from the usual arguments for GADTs, supporting GADTs has an additional benefit in a uniqueness type system. Consider the algebraic data type *Rose* of trees with an arbitrary number of branches. In *Clean*, this type is defined as

```
:: Rose a = Rose a [Rose a]
```

The problem with this definition is that it is unclear how the uniqueness of the list of rose trees relates to the uniqueness of the overall rose tree. Clean provides some hooks to influence this, but with a GADT, the problem disappears altogether since we can explicitly specify the type of the constructor:

data `Rose :: * -> *` **where**

$$\texttt{Rose} :: a^u \xrightarrow[\times]{u_f} \texttt{List}^v\,(\texttt{Rose}^v\,a^u) \xrightarrow[u]{u_{f'}} \texttt{Rose}^v\,a^u$$

With the definition as given, the list of rose trees must have the same uniqueness attribute as the overall rose tree (which can be accomplished in Clean by adding a dot, as in `.[Rose a]`), but other options are also possible[6].

The main problem with typing GADTs is that without type annotations, the type checker can no longer guarantee principal types (see [7] for an example). The solution is again to require type annotations, and distinguish between type inference and type checking.

6.3 Modifications to deal with uniqueness

In this section we briefly highlight how a type system based on [6] and [7] must be modified to deal with uniqueness, assuming that the starting point is the core uniqueness system from Section 3. Since we cannot give the full typing rules in this paper, we can only give intuitive descriptions in this section.

6.3.1 Abstractions

Recall from Section 3 that to be able to type a function, we must know whether the function will be able to extract unique elements from its closure. This is indicated by u_γ, and will be the case if the function is defined inside another function, and the argument to the outer function is unique. However, what if the argument to the outer lambda abstraction has a universally quantified uniqueness attribute?

$$\begin{aligned} &f :: (\forall u.a^u) \xrightarrow[\times]{u_f} \ldots \xrightarrow[?]{u_{f'}} \ldots \\ &f = \ldots \end{aligned}$$

What should the attribute at the location of the question mark be? We cannot simply use u, because u is not in scope. However, since the first argument to the function has a universally quantified uniqueness attribute, the function can assume at will that the argument is unique or non-unique (and pass it to another function that expects a unique argument); therefore, we should treat it as if the argument

[6]We do not require outwards propagation in the type of the constructor; it is possible to construct a unique rose tree with non-unique elements. This is impossible in Clean where the constructors enforce outwards propagation, but that is unnecessary. It suffices that the case statement enforces outwards propagation

had a unique attribute, and the attribute at the question mark should be $\bullet$. Thus, where in the core system we use the disjunction of u_γ and ν_1 to type the body of the function (where ν_1 is the attribute on the argument of the function), in the case of a lambda abstraction where the argument is annotated with a type scheme σ, we must use the disjunction of u_γ and $\lceil\sigma\rceil$, where the ceiling operator is defined as

$$\lceil\forall\overline{t},\overline{u}.\tau^\nu\rceil = \begin{cases} \nu & \text{if } \nu \notin \overline{u} \\ \bullet & \text{otherwise} \end{cases}$$

6.3.2 Recursive let definitions

For recursive let definitions, we follow the approach used in *Clean* [2], where a recursive let definition is always non-unique (since it points to itself). For annotated recursive let definitions, it is convenient to syntactically require that the annotation must be of the form $\forall\overline{t},\overline{u}.\tau^\times$ (i.e., the top-level attribute of the type scheme must be non-unique).

6.3.3 Case analysis

In [7], a number of rules are defined to typecheck the scrutinee of a case statement. In the most basic case there are two rules, one for atoms (variables and constructors) and one for all other expressions. These rules (and their more advanced variations) can be used without difficulty, but the rule for atoms must make sure to deal with sharing:

$$\mathsf{case}\ x^\otimes\ \mathsf{of}\ \ldots$$

Clearly, the scrutinee of a case expression must be given a non-unique type when it is marked as shared. The rules to type the branches of the case statement must get an additional premise that the attribute on the container must be the disjunction of the attributes on each of the elements of the container (see the discussion of *fst* in Section 4).

6.4 Complications due to inequalities

We argued above that it is easy to extend the core uniqueness system of this paper with advanced features such as arbitrary rank types and generalized algebraic data types. These extensions are 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 [1].

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 $\tau_a^{\nu_b}$ to a type scheme σ, $\tau_a^{\nu_a}$ 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 $\tau_b^{\nu_b}$, the same set of constraints should apply to $\tau_b^{\nu_b}$ as well. Thus, in [1], we defined a type scheme σ as

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

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 $\mathcal{C}$. The typing rules then are careful to manipulate constraint sets. For example, the rule for instantiating a type scheme read

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

This rule says that we can instantiate a type scheme to a type using a substitution $\mathcal{S}_x$, but we can only do so 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 (we could also modify the codomain, but that is not strictly necessary). 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 [1]; here we consider an example only. Suppose we have two functions f, g with types

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

$$g :: a^u \xrightarrow[u_a]{u_f} b^v, [u \leq v]$$

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

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

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

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

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.

The fact that we do not have to do anything special to define subsumption in this paper is interesting, and it is instructive to reconsider the last two examples. Recast in the new type system, the types of f and g are

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

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). 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'

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

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

We have shown that the major cause of the complexities of the types in our previous paper [1] is the presence of constraints. We have defined a uniqueness system for a core lambda calculus that is as expressive as our previous system, but does not require constraints anywhere. We claim that this makes the type system sufficiently similar to the Hindley/Milner type system that modern extensions can be added to it without much difficulty, and we have substantiated this claim by defining and implementing a uniqueness type system that supports arbitrary rank types and generalized algebraic data types. Most of the typing rules in this system are identical or very similar to their Hindley/Milner counterparts. Other extensions such as impredicativity should not be difficult to add either. We believe that we have designed a highly expressive uniqueness type system, that is practical to use and not difficult to understand.

ACKNOWLEDGEMENTS

We thank Dimitri Vytiniotis for his patient explanations of some of the details in his paper [7].

REFERENCES

[1] De Vries, E., Plasmeijer, R., Abrahamson, D.: Uniqueness typing redefined. In: Proceedings of the 18th International Symposium on Implementation and Application of Functional Languages. (2006) To be published; online at `http://www.cs.tcd.ie/∼devriese/pub`.

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

[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