Uniqueness Typing Redefined

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

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

Abstract. We modify *Clean*'s uniqueness type system in two ways. First, where in Clean functions that are partially applied to a unique argument are *necessarily* unique (they cannot lose their uniqueness), we just require that they must be unique *when applied*. This ultimately makes subtyping redundant. Second, we extend the type system to allow for higher rank types. To be able to do this, we explicitly associate type constraints (attribute inequalities) with type schemes. Consequently, types in our system are much more precise about constraint propagation.

1 Background

The problem of modelling side effects in pure functional languages, without losing referential transparency, is well-known. Consider the function `freadi` that reads the next integer from a file. The type of this function might be

```
freadi :: File → Int
```

To be able to return the *next* integer on every invocation, `freadi` advances the file pointer before returning. This side effect causes a loss of referential transparency. For instance, `f` and `g` are not interchangeable[3]:

```
f1 file = (freadi file) + (freadi file)
g1 file = (freadi file) ∗ 2
```

One way to make `freadi`'s side effect explicit is modifying its signature to

```
freadi :: World → File → (World, Int)
```

where `World` is some data type representing "the world". We must then redefine `f` and `g` as

```
f2 world file =
   let (world1, a) = freadi world file in
   let (world2, b) = freadi world1 file in
   (a + b, world2)
```

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

[3] The subscripts of f and g are used only to be able to refer to particular versions of f and g, and are not part of the code.

```
g2 world file =
   let (world1, a) = freadi world file in
   (a * 2, world1)
```

which makes it clear that `f` and `g` are in fact different functions. But the problem has not gone away, because nothing is stopping us from writing `f` as

```
f3 world file =
   let (world1, a) = freadi world file in
   let (world2, b) = freadi world file in
   (a + b, world2)
```

In the language *Haskell* this problem is essentially solved by hiding the "state threading" in a monad and never giving direct access to the `World` object. This makes programs "correct by construction", but rather affects the style of programming. By contrast, uniqueness typing enforces correct state threading in the type system. The main idea is to ensure that there is only ever a single (unique) reference to a particular world state. This is reflected in the type of `freadi`:

```
freadi :: World• → File → (World•, Int)
```

The bullets (•) indicate that `freadi` requires a unique reference to the `World`, and in turn promises to return a unique reference. When the compiler type-checks f_3, it finds that there are two references to `world`, which violates the uniqueness requirements; f_2 however is accepted.

The type system presented in this paper depends on a sharing analysis of the program, which is explained briefly in Sect. 2. Since the typing rules for rank-1 are easier to understand than the typing rules for arbitrary rank, we first present the rank-1 typing rules in Sect. 3 and then extend them to arbitrary rank in Sect. 4. We consider a few examples in Sect. 5, outline a type inference algorithm in Sect. 6, compare our system to the original Clean type system in Sect. 7, and present our conclusions and list future work in Sect. 8.

2 Sharing Analysis

The typing rules that we will present in this paper depend on a sharing analysis that marks variable uses as unique ($\odot$) or non-unique ($\otimes$). 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 non-unique. In this paper, we assume sharing analysis has been done, leaving a formal definition to future work. Here we look at an example only. Compare again the definitions of f_2 and f_3 from Sect. 1. In the correct definition (f_2), the variable marking indicates that the reference to `world` is indeed unique (as required by `freadi`):

```
f2 world file =
   let (world1, a) = freadi⊗ world⊙ file⊗ in
   let (world2, b) = freadi⊗ world1⊙ file⊗ in
   (a⊙ + b⊙, world2⊙)
```

The marking in the incorrect definition indicates that there is more than one reference to the same world state, violating the uniqueness requirement:

```
f3 world file =
   let (world1, a) = freadi⊗ world⊗ file⊗ in
   let (world2, b) = freadi⊗ world⊗ file⊗ in
   (a⊙ + b⊙, world2⊙)
```

In Sect. 5, we will look at an example that can be typed only if a more sophisticated sharing analysis is applied.

3 Introducing Uniqueness Typing

We will present a uniqueness type system that allows for rank-1 types only, before showing the full type system in Sect. 4. Although both the expression language and the type language must be modified to support arbitrary rank types, the typing rules as presented in this section are easier to understand and are therefore a better way to introduce the type system.

3.1 The Language

We define our type system over a core lambda calculus[4]:

$e ::=$		expression
	$x^{\odot}$, $x^{\otimes}$	variable
	$\lambda x \cdot e$	abstraction
	$e_1\, e_2$	application
	i	integer

The typing rules assign an attributed type τ^{ν} to an expression e, given a type environment Γ and a uniqueness attribute u_γ (explained in Sect. 3.4), denoted

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

The language of types and uniqueness attributes is defined as

$\tau ::=$		type	$\nu ::=$		uniqueness attribute
	a, b	type variable		u, v	variable
	$\tau_1^{\nu_1} \underset{\nu_a}{\longrightarrow} \tau_2^{\nu_2}$	function		$\bullet$	unique
	Int	constant type		$\times$	non-unique

The syntax for arrows warrants a closer look. The domain and codomain of the arrow are two attributed types $\tau_1^{\nu_1}$ and $\tau_2^{\nu_2}$. The arrow itself has an *additional*

[4] Although we could include `let` definitions, the corresponding typing rule would look quite different from the typing rule that we will eventually use, so there is no point in including it here. We will give a typing rule for `let` definitions in Sect. 4.

attribute ν_a, whose role will be become apparent when we discuss the rule for abstractions. We will adopt the notational convention of writing $(\tau_1^{\nu_1} \underset{\nu_a}{\longrightarrow} \tau_2^{\nu_2})^{\nu_f}$, where ν_f is "normal" uniqueness attribute of the arrow, as $(\tau_1^{\nu_1} \xrightarrow[\nu_a]{\nu_f} \tau_2^{\nu_2})$.

As is customary, all type and attribute variables in an attributed type τ^ν are implicitly universally quantified at the outermost level (of course, that will not be true for the arbitrary rank system). In this section, a type environment maps variable names to attributed types (in Sect. 4, it will map variable names to type schemes).

3.2 Integers

We can specify two alternative rules for integers:

$$\frac{}{\Gamma, u_\gamma \vdash i : \mathrm{Int}^\nu}\ \textsc{Int} \qquad \frac{}{\Gamma, u_\gamma \vdash i : \mathrm{Int}^\bullet}\ \textsc{Int}'$$

Int says that integers have type Int^ν, for an arbitrary ν: the programmer is free to assume the integer is unique or non-unique. Alternatively, Int$'$ states that an integer is always unique. We will discuss why we prefer Int in Sect. 3.4.

3.3 Variables

The rule for variables that are marked unique (Var$^\odot$) simply states that to find the type of the variable, we look up the variable in the environment. For variables that are marked non-unique, we do the same, but then correct the type to be non-unique:

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

Note that Var$^\otimes$ leaves the uniqueness attribute of the variable *in the environment* arbitrary. This means that variables can "lose" their uniqueness. For example, the function `mkPair` defined as $\lambda x \cdot (x^\otimes, x^\otimes)$ has type $a^u \to (a^\times, a^\times)$ (assuming a product type); in other words, no matter what the uniqueness of a on input is, the as in the pair will be non-unique.

3.4 Abstractions

Before we discuss the typing rule for abstractions, we must return to the example discussed in Sect. 1 and point out a subtlety. Consider f_3 again:

```
f3 world file =
   let (world1, a) = freadi⊗ world⊗ file⊗ in
   let (world2, b) = freadi⊗ world⊗ file⊗ in
   (a⊙ + b⊙, world2⊙)
```

The compiler is able to reject this definition because `world` is marked as non-unique, which will cause its type to be inferred as non-unique by rule VAR$^{\otimes}$. But what happens if we "curry" `freadi`?

```
f world file =
   let curried = freadi⊙ world⊙ in
   let (world1, a) = curried⊗ file⊗ in
   let (world2, b) = curried⊗ file⊗ in
   (a⊙ + b⊙, world2⊙)
```

Both programs are semantically equivalent, so the type-checker should reject both programs. However, the argument `world` to `freadi` is in fact unique in the second example, so how can we detect the type error? The general principle is

> *when a function accesses unique objects from its closure, that closure (i.e., the function) must itself be unique* $(*)$

In the example above, `curried` accesses a unique file handle from its closure, and must therefore itself be unique—but is not, resulting in a type error. We can approximate[5] $(*)$ by

> *if a function is curried, and the curried argument is unique, the result function must be unique when applied* $(*')$

In the lambda calculus, functions only ever have a single argument, and the notion of currying translates into lambda abstractions returning new lambda abstractions. Thus, we can rephrase $(*')$ as

> *if a lambda abstraction returns a new lambda abstraction, and the argument to the outer lambda abstraction is unique, the inner lambda abstraction must be unique when applied* $(*'')$

In our type language, the additional attribute ν_a in the arrow type $\tau_1^{\nu_1} \underset{\nu_a}{\longrightarrow} \tau_2^{\nu_2}$ indicates whether the function is required to be "unique when applied". The purpose of u_γ in the typing rules is to indicate whether we are currently in the body of an (outer) lambda abstraction whose argument must be unique. Thus we arrive at rule ABS:

$$\frac{(\Gamma, x : \tau_1^{\nu_1}), u_{\gamma'} \vdash e : \tau_2^{\nu_2} \qquad \nu_a \leq u_\gamma, u_{\gamma'} \leq \nu_1, u_{\gamma'} \leq u_\gamma}{\Gamma, u_\gamma \vdash \lambda x \cdot e : \tau_1^{\nu_1} \underset{\nu_a}{\overset{\nu_f}{\longrightarrow}} \tau_2^{\nu_2}} \quad \text{ABS}$$

This rule is very similar to the normal rule for abstractions in a Hindley/Milner type system, with the exception of the attribute inequalities in the premise of the rule. The $u \leq v$ operator can be read as an implication: if v is unique, then u must be unique (v implies u, $u \leftarrow v$).

[5] This is an approximation since the function may not use the curried argument. In $\lambda x \cdot \lambda y \cdot y^{\odot}$, x is not used in the body of the function, so that its uniqueness need not affect the type of the function.

The first constraint establishes the conclusion of $(*'')$: if we are in the body of an outer lambda abstraction whose argument must be unique (u_γ), then the inner lambda abstraction must be unique when applied (ν_a). The second constraint $u_{\gamma'} \leq \nu_1$ is a near direct translation of the premise of $(*'')$. Finally, $u_{\gamma'} \leq u_\gamma$ simply propagates u_γ: if the premise of $(*'')$ already holds (u_γ), it will continue to do so in the body of the abstraction ($u_{\gamma'}$). Note that Abs is the only rule that changes the value of u_γ; all the other rules simply propagate it. When typing an expression, u_γ is initially assumed to be non-unique.

It is probably instructive to consider an example at this point. We show the type derivation for $\lambda x \cdot \lambda y \cdot x^{\odot}$, the function that returns the first of its two arguments:

$$\dfrac{\dfrac{\dfrac{}{(x : \tau_1^{\nu_1}, y : \tau_2^{\nu_2}), u_{\gamma''} \vdash x^{\odot} :: \tau_1^{\nu_1}} \textsc{Var}^{\odot} \qquad \nu_{a'} \leq u_{\gamma'}, u_{\gamma''} \leq \nu_2, u_{\gamma''} \leq u_{\gamma'}}{(x : \tau_1^{\nu_1}), u_{\gamma'} \vdash \lambda y \cdot x^{\odot} :: \tau_2^{\nu_2} \xrightarrow[\nu_{a'}]{\nu_{f'}} \tau_1^{\nu_1}} \textsc{Abs} \qquad \nu_a \leq \times, u_{\gamma'} \leq \nu_1, u_{\gamma'} \leq \times}{\emptyset, \times \vdash \lambda x \cdot \lambda y \cdot x^{\odot} :: \tau_1^{\nu_1} \xrightarrow[\nu_a]{\nu_f} (\tau_2^{\nu_2} \xrightarrow[\nu_{a'}]{\nu_{f'}} \tau_1^{\nu_1})} \textsc{Abs}$$

Noting that $\nu_a \leq \times$ and $u_{\gamma'} \leq \times$ are vacuously true, that $u_{\gamma''} \leq \nu_2$ and $u_{\gamma''} \leq u_{\gamma'}$ are irrelevant because $u_{\gamma''}$ is never referenced, and that $\nu_{a'} \leq u_{\gamma'}$ and $u_{\gamma'} \leq \nu_1$ imply that $\nu_{a'} \leq \nu_1$ (by transitivity), we arrive at the type

$$\lambda x \cdot \lambda y \cdot x^{\odot} :: \tau_1^{\nu_1} \xrightarrow[\nu_a]{\nu_f} (\tau_2^{\nu_2} \xrightarrow[\nu_{a'}]{\nu_{f'}} \tau_1^{\nu_1}) \qquad \nu_{a'} \leq \nu_1$$

where the constraint $\nu_{a'} \leq \nu_1$ says that if we curry the function (specify x but not y), and x happens to be unique, the result function must be unique on application (its attribute $\nu_{a'}$ must be $\bullet$).

If we now consider rule Int′, which says that integers are always unique, this definition of Abs would imply that if we curry a function by passing in an integer, the result function must be unique on application, which is unnecessary. For example, we want the following expression to be type correct:

$$\text{let fst} = \lambda x \cdot \lambda y \cdot x \text{ in let one} = \text{fst } 1 \text{ in (one } 2, \text{one } 3)$$

For the same reason, nothing in Abs constrains ν_f: the actual uniqueness of the function is left free.

3.5 Application

The rule for function application is relatively straightforward. The only difference between the rule as presented here and the usual definition is that App enforces the constraint that functions that must be unique when applied, are in fact unique when applied ($\nu_f \leq \nu_a$):

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

4 Arbitrary Rank Types

In the rank-1 system (as well as in Clean's type system), constraints are never explicitly associated with types, but are left implicit in the typing rules. Although that makes the types simpler, we can no longer do so if we want to support arbitrary rank types. When we generalise a type τ^ν to a type scheme σ, τ^ν 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 $\tau^{\nu\prime}$, the same set of constraints should apply to $\tau^{\nu\prime}$ as well. Although this makes the types more complicated, it also makes them more precise (see also sections 7 and 8). So, we define a type scheme as

$\sigma ::=$	$\forall \vec{x}.\tau^\nu, \mathcal{C}$	type scheme

Type inference for arbitrary rank types is undecidable, but it is possible to combine type inference with type checking and allow for higher rank types only if lambda arguments have an explicit type annotation [2]. We extend the expression language with annotated lambda expressions (and `let` expressions):

$e \mathrel{+}=$		expression *(ctd.)*
	$\lambda x :: \sigma \cdot e$	annotated abstraction
	let $x = e$ in e'	local definition

We modify the type language to allow for type schemes in the domain of the arrow type. We follow [3] and do not allow for type schemes in the codomain:

$\tau ::=$		type
	a, b	type variable
	$\sigma \underset{\nu_a}{\longrightarrow} \tau_2^{\nu_2}$	arrow type (functions)
	Int	constant type

Typing derivations now have the structure

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

which says that e has type τ^ν, given an environment Γ and uniqueness attribute u_γ (see Sect. 3.4), provided constraints $\mathcal{C}$ are satisfied (where environments now map variable names to type schemes). The full typing rules are listed in Fig. 1; we will explain them separately below.

4.1 Variables

Because the type environment now associates variable names with type schemes rather than types, to find the type of a variable we must lookup the associated type scheme in the environment, and instantiate it. Instantiation is defined as

$$\frac{}{\vdash^{\text{inst}} \forall \vec{x}.\tau^\nu, \mathcal{C} \preceq \mathcal{S}_x\tau^\nu \,|\, \mathcal{S}_x\mathcal{C}} \quad \textsc{Inst}$$

$$\frac{}{\Gamma, u_\gamma \vdash i : \mathrm{Int}^\nu \,|\, \emptyset} \quad \textsc{Int}$$

$$\frac{\vdash^{\mathrm{inst}} \sigma \preceq \tau^\nu \,|\, \mathcal{C}}{(\Gamma, x : \sigma), u_\gamma \vdash x^\odot : \tau^\nu \,|\, \mathcal{C}} \quad \textsc{Var}^\odot$$

$$\frac{\vdash^{\mathrm{inst}} \sigma \preceq \tau^\nu \,|\, \mathcal{C}}{(\Gamma, x : \sigma), u_\gamma \vdash x^\otimes : \tau^\times \,|\, \mathcal{C}} \quad \textsc{Var}^\otimes$$

$$\frac{(\Gamma, x : \forall.\tau_1^{\nu_1}, \mathcal{C}_1), u_{\gamma'} \vdash e : \tau_2^{\nu_2} \,|\, \mathcal{C}_2}{\Gamma, u_\gamma \vdash \lambda x \cdot e : (\forall.\tau_1^{\nu_1}, \mathcal{C}_1) \xrightarrow[\nu_a]{\nu_f} \tau_2^{\nu_2} \,|\, \mathcal{C}_2, \nu_a \leq u_\gamma, u_{\gamma'} \leq u_\gamma, u_{\gamma'} \leq \nu_1} \quad \textsc{Abs}$$

$$\frac{\Gamma, u_\gamma \vdash e_1 : \sigma_1 \xrightarrow[\nu_a]{\nu_f} \tau_2^{\nu_2} \,|\, \mathcal{C} \qquad \Gamma, u_\gamma \vdash^{\mathrm{gen}} e_2 : \sigma_2 \qquad \vdash^{\mathrm{subs}} \sigma_2 \preceq \sigma_1}{\Gamma, u_\gamma \vdash e_1\, e_2 : \tau_2^{\nu_2} \,|\, \mathcal{C}, \nu_f \leq \nu_a} \quad \textsc{App}$$

$$\frac{\Gamma, u_\gamma \vdash^{\mathrm{gen}} e : \sigma \qquad (\Gamma, x : \sigma), u_\gamma \vdash e' : \tau^\nu \,|\, \mathcal{C}}{\Gamma, u_\gamma \vdash \mathrm{let}\ x = e\ \mathrm{in}\ e' : \tau^\nu \,|\, \mathcal{C}} \quad \textsc{Let}$$

$$\frac{(\Gamma, x : \sigma), u_{\gamma'} \vdash e : \tau_2^{\nu_2} \,|\, \mathcal{C}}{\Gamma, u_\gamma \vdash \lambda x :: \sigma \cdot e : \sigma \xrightarrow[\nu_a]{\nu_f} \tau_2^{\nu_2} \,|\, \mathcal{C}, \nu_a \leq u_\gamma, u_{\gamma'} \leq u_\gamma, u_{\gamma'} \leq \lceil \sigma \rceil} \quad \textsc{Annot}$$

$$\frac{\Gamma, u_\gamma \vdash e : \tau^\nu \,|\, \mathcal{C} \qquad \vec{x} = \mathrm{freevars}(\tau^\nu) - \mathrm{freevars}(\Gamma)}{\Gamma, u_\gamma \vdash^{\mathrm{gen}} e : \forall \vec{x}.\tau^\nu, \mathcal{C}} \quad \textsc{Gen}$$

$$\frac{}{\vdash^{\mathrm{inst}} \forall \vec{x}.\tau^\nu, \mathcal{C} \preceq \mathcal{S}_x \tau^\nu \,|\, \mathcal{S}_x \mathcal{C}} \quad \textsc{Inst}$$

$$\frac{\vec{y} \notin \mathrm{freevars}(\forall \vec{x}.\tau_1^{\nu_1}) \qquad \vdash^{\mathrm{subs}} S_x \tau_1^{\nu_1} \preceq \tau_2^{\nu_2} \qquad C_2 \vDash \mathcal{S}_x C_1}{\vdash^{\mathrm{subs}} \forall \vec{x}.\tau_1^{\nu_1}, \mathcal{C}_1 \preceq \forall \vec{y}.\tau_2^{\nu_2}, \mathcal{C}_2} \quad \textsc{Subs}^\sigma$$

$$\frac{\vdash^{\mathrm{subs}} \sigma_2 \preceq \sigma_1 \qquad \vdash^{\mathrm{subs}} \forall.\tau_1^{\nu_1}, \emptyset \preceq \forall.\tau_2^{\nu_2}, \emptyset}{\vdash^{\mathrm{subs}} \sigma_1 \to \tau_1^{\nu_1} \preceq \sigma_2 \to \tau_2^{\nu_2}} \quad \textsc{Subs}^\to$$

$$\frac{}{\vdash^{\mathrm{subs}} \tau^\nu \preceq \tau^\nu} \quad \textsc{Subs}^\tau$$

Fig. 1. Uniqueness Typing Rules

where S_x is some substitution $[\vec{x} \mapsto \ldots]$ mapping all variables $\vec{x}$ to fresh variables. Since we associate a set of constraints $\mathcal{C}$ with a type scheme, a type $S_x\tau^\nu$ is only an instance of a type scheme σ if those constraints are satisfied.

4.2 Abstraction

The rule for abstraction remains unchanged except for the fact that the domain of the arrow operator is now a type scheme. However, since we can only infer rank-1 types, the type scheme for annotated lambda expressions must be a "degenerate" type scheme with no quantified variables $(\forall.\tau^\nu, \mathcal{C})$—in other words, a type[6].

4.3 Application

The rule for application does look slightly different from the rank-1 version. Previously, App required that the type of the actual parameter must equal the type of the formal parameter of the function:

$$\frac{\Gamma, u_\gamma \vdash e_1 : \tau_1^{\nu_1} \xrightarrow[\nu_a]{\nu_f} \tau_2^{\nu_2} \qquad \Gamma, u_\gamma \vdash e_2 : \tau_1^{\nu_1} \qquad \nu_f \leq \nu_a}{\Gamma, u_\gamma \vdash e_1\, e_2 : \tau_2^{\nu_2}} \quad \textsc{App}_1$$

In the rank-n case, the only requirement is that the type of the actual parameter is an instance of the type of the formal parameter. To this end, we infer a type scheme for the actual parameter, and do a subsumption check:

$$\frac{\Gamma, u_\gamma \vdash e_1 : \sigma_1 \xrightarrow[\nu_a]{\nu_f} \tau_2^{\nu_2} \,|\, \mathcal{C} \qquad \Gamma, u_\gamma \vdash^{\text{gen}} e_2 : \sigma_2 \qquad \vdash^{\text{subs}} \sigma_2 \preceq \sigma_1}{\Gamma, u_\gamma \vdash e_1\, e_2 : \tau_2^{\nu_2} \,|\, \mathcal{C}, \nu_f \leq \nu_a} \quad \textsc{App}$$

(We will explain subsumption separately in section 4.5.) To infer a type scheme, we first infer a type, and then generalise over all the free variables in the type, excluding the free variables in the environment:

$$\frac{\Gamma, u_\gamma \vdash e : \tau^\nu \,|\, \mathcal{C} \qquad \vec{x} = \text{freevars}(\tau^\nu) - \text{freevars}(\Gamma)}{\Gamma, u_\gamma \vdash^{\text{gen}} e : \forall\vec{x}.\tau^\nu, \mathcal{C}} \quad \textsc{Gen}$$

4.4 Annotated Lambda Abstractions

The rule for annotated lambda abstractions is very similar to the rule for "ordinary" lambda abstractions, except that the programmer can now specify a type scheme manually, allowing for higher rank types:

$$\frac{(\Gamma, x : \sigma), u_{\gamma'} \vdash e : \tau_2^{\nu_2} \,|\, \mathcal{C}}{\Gamma, u_\gamma \vdash \lambda x :: \sigma \cdot e : \sigma \xrightarrow[\nu_a]{\nu_f} \tau_2^{\nu_2} \,|\, \mathcal{C}, \nu_a \leq u_\gamma, u_{\gamma'} \leq u_\gamma, u_{\gamma'} \leq \lceil\sigma\rceil} \quad \textsc{Annot}$$

[6] In [3] the arrow $\rightarrow$ is overloaded; there is an arrow $\tau \rightarrow \tau$ and an arrow $\sigma \rightarrow \tau$. Since we do not use the notion of ρ–types, our arrows always have type $\sigma \rightarrow \tau^\nu$.

We have to be careful defining $\lceil \forall \vec{x}.\tau^{\nu} \rceil$. The obvious answer ($\nu$) is only correct if ν is not itself universally quantified. For example, consider

$$\lambda x :: \forall u \cdot a^u . \lambda y . x^{\odot} :: (\forall u \cdot a^u) \xrightarrow[u_a]{u_f} b^v \xrightarrow[u_{a'}]{u_{f'}} a^u, ?$$

(Note that this is a rank-2 type.) The question is what the constraint at the question mark should be. One possible solution is

$$\forall u \cdot u_{a'} \leq u$$

but that is equivalent to saying

$$u_{a'} \leq \bullet$$

So, to avoid unnecessary complication by introducing universal quantification into the constraint language, we define $\lceil \rceil$ as

$$\lceil \forall \vec{x}.\tau^{\nu} \rceil = \begin{cases} \nu & \text{if } \nu \notin \vec{x} \\ \bullet & \text{otherwise} \end{cases}$$

4.5 Subsumption

The rules for subsumption are defined nearly exactly as in [3], except that we have collapsed rules Skol and Spec into one rule (Subs$^{\sigma}$) and added one additional premise. Subs$^{\sigma}$ is the main rule that checks whether one type scheme is a (generic) instance of another:

$$\frac{\vec{y} \notin \text{freevars}(\forall \vec{x}.\tau_1^{\nu_1}) \qquad \vdash^{\text{subs}} S_x \tau_1^{\nu_1} \preceq \tau_2^{\nu_2} \qquad C_2 \vDash \mathcal{S}_x C_1}{\vdash^{\text{subs}} \forall \vec{x}.\tau_1^{\nu_1}, \mathcal{C}_1 \preceq \forall \vec{y}.\tau_2^{\nu_2}, \mathcal{C}_2} \quad \text{Subs}^{\sigma}$$

The constraint $\mathcal{C}_2 \vDash S_x \mathcal{C}_1$ is new, and is probably best explained by example. Suppose we have two functions f, g with types

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

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! Thus, the application $f\ g$ should be disallowed. Conversely, if we instead define f' and g' as

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

the application $f'\,g'$ *should* be allowed because the type of g' is more general than the type expected by f'. The condition $\mathcal{C}_2 \vDash S_x\mathcal{C}_1$, where the $\vDash$ symbol is logical entailment from propositional logic, says that if constraints $\mathcal{C}_2$ are satisfied, constraints $\mathcal{C}_1$ must also be satisfied[7]. In other words, the constraints of the offered type must be the same or less restrictive than the constraints of the requested type.

5 Examples

In this section we consider a few example expressions and their associated types. We start with very simple expressions and slowly build up from there. First, we consider a single integer:

$$5 :: \forall u.\mathrm{Int}^u, \emptyset$$

Rule Int says that integers have type Int with an arbitrary uniqueness, hence the universally quantified u. Next we consider the identity function `id`:

$$\lambda x.x^{\odot} :: \forall a, u, u_f, u_a, c.(\forall.a^u, c) \xrightarrow[u_a]{u_f} a^u, c$$

This type may look a bit more complicated than it really is, because we show top-level attributes and degenerate type schemes. If we are slightly less formal:

$$\lambda x.x^{\odot} :: (a^u, c) \xrightarrow[u_a]{u_f} a^u, c$$

Either way, this is the type one would expect an identity function to have. Note that this function is polymorphic in the constraints of its argument: if the argument has type a^u under constraints c, then the result has type a^u only if the same set of constraints is satisfied.

The function `apply` ($ in Haskell) behaves like the identity function restricted to function types:

$$\lambda f.\lambda x.f^{\odot}\,x^{\odot} :: \left((a^u, c_1) \xrightarrow[u_{a''}]{u_{f''}} b^v, c_2\right) \xrightarrow[u_a]{u_f} \left((a^u, c_1) \xrightarrow[u_{a'}]{u_{f'}} b^v\right), [c_2, u_{a'} \leq u_{a''}, u_{a'} \leq u_{f''}, u_{f''} \leq u_{a''}]$$

This type of `apply` should be self-explanatory, with the exception perhaps of the constraints. We consider each constraint in turn:

[7] If either $\mathcal{C}_1$ or $\mathcal{C}_2$ in $\mathcal{C}_1 \vDash \mathcal{C}_2$ is a constraint variable, we apply unification instead of the entailment check.

c_2	If f has type $(a^u, c_1) \xrightarrow[u_{a''}]{u_{f''}} b^v$ only when constraints c_2 are satisfied, then `apply` f also has that type only when those constraints are satisfied (*cf.* the constraint c in the type of `id`.)
$u_{a'} \leq u_{a''}$	If f can only be executed once (in other words, if f must be unique on application, if $u_{a''}$ is unique), then `apply` f can also only be executed once.
$u_{a'} \leq u_{f''}$	If f is unique, then `apply` f can only be executed once; this is a direct consequence of the "currying rule" from Sect. 3.4.
$u_{f''} \leq u_{a''}$	Finally, `apply` f does actually apply f, so if f must be unique on application, we require that it is in fact unique.

The next example emphasises a point with respect to the sharing analysis (the marking of variable uses). Suppose that we have a primitive type Array and two functions *resize* to (destructively) resize the array, and *size* to return the current size of the array:

$$\begin{aligned} &\mathit{resize} :: \mathrm{Array}^{\bullet} \xrightarrow[u_a]{u_f} \mathrm{Int}^v \xrightarrow[\bullet]{u_{f'}} \mathrm{Array}^{\bullet} \\ &\mathit{size} :: \mathrm{Array}^u \xrightarrow[u_a]{u_f} \mathrm{Int}^v \end{aligned}$$

Then the following expression is correctly marked and type correct:

$$\lambda \mathit{arr} \cdot \text{if } \mathit{size}^{\odot}\ \mathit{arr}^{\otimes} < 10 \text{ then } \mathit{resize}^{\otimes}\ \mathit{arr}^{\odot}\ 20 \text{ else } \mathit{resize}^{\otimes}\ \mathit{arr}^{\odot}\ 30$$

This expression is marked correctly, because only one of the two branches of the conditional will be executed, and the non-unique mark $\mathit{arr}^{\otimes}$ in the condition guarantees that the condition cannot modify *arr*.

To conclude this section, we consider two examples that contain a type error, which in both cases will be detected in the subsumption check (although for different reasons). In the first example, it is a simple case of an argument not being polymorphic enough:

$$\begin{aligned} &\text{let } \mathit{id}_{\mathrm{f}} = \lambda f :: \forall u.a^u \xrightarrow[u_a]{u_f} a^u \cdot f^{\odot} \\ &\text{in let } \mathit{id}_{\mathrm{int}} = \lambda i :: \mathrm{Int}^{\bullet} \cdot i^{\odot} \\ &\text{in } \mathit{id}_{\mathrm{f}}^{\odot}\ \mathit{id}_{\mathrm{int}}^{\odot} \end{aligned}$$

In this example, id_{f} demands that its argument is polymorphic in u, but $\mathit{id}_{\mathrm{int}}$ is not (in fact, works only on unique integers). So, the type-checker will give an error message similar to

`Cannot unify rigid attribute` u `and` $\bullet$

The second "wrong" example that we consider fails due to the entailment check explained in section 4.5:

$$\begin{aligned}&\text{let } \mathit{first} = \lambda f :: a^u \xrightarrow[u_a]{u_f} b^v \xrightarrow[u_{a'}]{u_{f'}} a^u \cdot \lambda x \cdot \lambda y \cdot f^{\odot}\, x^{\odot}\, y^{\odot}\\ &\text{in } \mathit{first}^{\odot}\, (\lambda x \cdot \lambda y \cdot x^{\odot})\end{aligned}$$

Function *first* returns the first of two arguments x and y, but it delegates that task to a function f, which must also be passed in. However, the function that is passed in has type[8]

$$\lambda x \cdot \lambda y \cdot x^{\odot} :: a^u \xrightarrow[u_a]{u_f} b^v \xrightarrow[u_{a'}]{u_{f'}} a^u, [u_{a'} \leq u]$$

whereas the type specified for the argument f of *first* does not allow for the constraint $u_{a'} \leq u$; so, the type-checker will fail with

`[] does not entail` $[u_{a'} \leq u]$

6 Type Inference

We have written a prototype implementation of the type system presented in this paper. The typing rules as presented in Fig. 1 allow for a relatively straightforward translation to an algorithm $\mathcal{W}$ [4] style type-checker (our prototype is just under a thousand lines long), once the following subtleties are observed.

When doing unification, a unification goal, $\tau_1^{\nu_1} \equiv \tau_2^{\nu_2}$ should be expanded into two subgoals $\tau_1 \equiv \tau_2$ and $\nu_1 \equiv \nu_2$. In other words, the base types and the uniqueness attributes should be unified independently.

Unification should not be used to unify functions, because as far as unification is concerned, $\sigma_1 \to \tau_1^{\nu_1} \equiv \sigma_2 \to \tau_2^{\nu_2}$ is the same as $\sigma_2 \to \tau_2^{\nu_2} \equiv \sigma_1 \to \tau_1^{\nu_1}$, but to compare two type schemes we need to use subsumption, which obviously *does* give different answers for $\vdash^{\text{subs}} \sigma_1 \preceq \sigma_2$ and $\vdash^{\text{subs}} \sigma_2 \preceq \sigma_1$. However, when implemented properly, by the time we need unification, the subsumption rules will have taken care of all arrows[9].

To implement the subsumption check, the technique suggested by Peyton Jones [3] of using skolem constants can be used without modification; all one has to do is to introduce skolem constants for the uniqueness attributes too (these are "rigid attributes" in the type error in Sect. 5).

Logical entailment of two sets of constraints $\mathcal{C}_1$ and $\mathcal{C}_2$ can be implemented as a validity check for the propositional logic formula $C_1 \to C_2$, where the $u \leq v$ operator is regarded as an implication $v \to u$. Although the complexity of

[8] There are additional "polymorphic" constraint variables in these types that we are leaving out for conciseness.

[9] In [3], due to the distinction between ρ functions and τ functions, unification must still deal with arrows $\tau \to \tau$; since we only have one arrow type, this is unnecessary in our approach.

checking the validity of functions in propositional logic is exponential, that will not matter much in practice as the formulae generated by the type-checker will be small. See [5, Sect. 1.5] for a simple algorithm.

Finally, when generalising a type τ^ν with respect to a set of constraints $\mathcal{C}$, that set should be checked for inconsistencies, which should be reported as type errors. For readability of the types, it is also useful to take the transitive closure of $\mathcal{C}$ instead of $\mathcal{C}$ itself, and only add the "relevant" inequalities to the type scheme (rule Abs might generate unnecessary constraints $[u_{\gamma'} \leq u_\gamma, u_{\gamma'} \leq \nu_1]$ if $u_{\gamma'}$ is never used in the body of the abstraction); this is demonstrated in the example in Sect. 3.4.

7 Comparison with Clean

The uniqueness type system presented here is based on the uniqueness type system of the functional programming language *Clean* [1, 6], which is in turn strongly related to substructural logics (see [7] for an accessible introduction). There are however a number of important differences. The first obvious difference is that Clean's system is defined over graph rewrite rules rather than the lambda calculus; this gives the type system a very different "feel".

A rather more important difference is the treatment of curried functions. In Clean, a function that is (partially) applied to a unique argument, is itself unique. Moreover, unique functions are *necessarily unique*: they cannot lose their uniqueness. In the curry example in Sect. 3.4, there are two references to `curried`, causing `curried` to be marked as $\otimes$. The type correction in rule $\textsc{Var}^\otimes$ (a trivial operation in our system) must then check whether the variable in fact represents a function, and if so, reject the program. While this does solve the curried function problem, it has far reaching consequences for the type system.

The first consequence is that type variables are not allowed to lose their uniqueness either, since a type variable can be instantiated to a function type. For example, in Clean, the function `mkPair` has type

$$\lambda x \cdot (x^\otimes, x^\otimes) :: a^\times \to (a^\times, a^\times)$$

instead of

$$\lambda x \cdot (x^\otimes, x^\otimes) :: a^u \to (a^\times, a^\times)$$

The type assigned by Clean is not as restrictive at is seems, however, due to Clean's subtyping relation: a unique type is considered to be *subtype* of its non-unique counterpart. For example, the following is a correct Clean program:

```
five :: Int•
five = 5

mkPair :: a× → (a×, a×)
mkPair x = (x, x)

Start = mkPair five
```

where `Start` is assigned the type ($\text{Int}^{\times}$, $\text{Int}^{\times}$). Of course, the subtyping relation is adapted for arrows [6]:

$$S \xrightarrow{u} S' \leq T \xrightarrow{v} T' \qquad \text{iff} \qquad u = v \text{ and } T \leq S \text{ and } S' \leq T'$$

There are two things to note about this definition. First of all, a unique function is never a subtype of its non-unique version (condition $u = v$), since functions are not allowed to lose their uniqueness (a similar restriction applies to type variables). The second thing to note is that the subtyping is contravariant in the function argument. Although that is not surprising, it complicates the type system, especially in the presence of algebraic data types. We have not discussed algebraic data types at all in this paper (see Sect. 8), but they are easy to add to our system. However, algebraic data constructors can include arrows, for example

```
data Fun a b = Fun (a → b)
```

which means that arguments to constructors must be analysed to check whether they have covariant, contravariant or invariant subtyping behaviour.

By contrast, in our system we do not have the notion of "necessarily unique"; instead, we add a single additional attribute ν_a as explained before, and the condition that (some) curried functions can only be executed once becomes a local constraint $\nu_f \leq \nu_a$ in the rule for function application. There are no global effects (for example, type variables are unaffected) and we do not need subtyping[10].

That last point is worth emphasising. The subtyping relation in Clean is very shallow. The only advantage of subtyping is that we can pass in a unique object to a function that expects a non-unique object. So, in Clean, marking a formal parameter as non-unique really means, "I do not care about the uniqueness of this parameter". However, in our system, we can always use an attribute variable to mean the same thing. That is not always possible in Clean, since type variables are not allowed to lose their uniqueness (the type we assign to the function `mkPair` above would be illegal in Clean).

Since we do not have subtyping, functions can require their arguments to be unique ($a^{\bullet}$), non-unique ($a^{\times}$), or indicate that the uniqueness of the input does not matter (a^{u}). In Clean, it is only possible to require that arguments are unique ($a^{\bullet}$) or that the uniqueness of the attribute does not matter (a^{u} or, due to subtyping, $a^{\times}$). Experience will tell whether this extra functionality is useful.

Finally, [6, p. 30, Sect. *Uniqueness Type Inference*] states:

> *However, because of our treatment of higher-order functions (involving a restricting on the subtype relation w.r.t variables), it might be the case that lifting this most general solution fails, whereas some specific instance is attributable. (...) Therefore, there is no "Principal Uniqueness Type Theorem".*

[10] One might argue that subsumption introduces subtyping between type schemes; however, due to the predicative nature of our type system, this does not have an effect on algebraic data type arguments; see the discussion in [3, Sect. 7.3].

Although a formal proof is future work, the authors hope that the system presented here *does* have principal types.

Finally, the original motivation for this work was the fact that Clean's uniqueness system does not allow for arbitrary rank types. An additional benefit of allowing for type schemes in the domain of arrows (necessary to support higher rank types) is the fact that we can be more conscientious about associating uniqueness inequalities (constraints) with types. For example, in Clean, the function `apply` from Sect. 5 has type

$$\lambda f \cdot \lambda x \cdot f\, x :: (a^u \to b^v) \to a^u \to b^v$$

But given a function f with type

$$f :: a^u \to b^v, [u \leq v]$$

the Clean type-checker assigns the following type to $\texttt{apply}\, f$:

$$\texttt{apply}\, f :: a^u \to b^v, [u \leq v]$$

That type is quite reasonable, and in fact very similar to the type we would assign. It does however contain constraints that do not appear in the type of `apply`, which suggests that the type of `apply` as assigned by the Clean type-checker is somehow "incomplete". The type we assign to `apply` is very explicit about the propagation of constraints (leaving out the attributes on the arrows):

$$\lambda f \cdot \lambda x \cdot f\, x :: ((a^u, c_1) \to b^v, c_2) \to (a^u, c_1) \to b^v, c_2$$

8 Future Work and Conclusions

We have designed a uniqueness type system for the lambda calculus that can be used to add side effects to a pure functional language without losing referential transparency. This type system is based on the type system of the functional programming language *Clean*, but modifies it in a number of ways. First, it is defined over the lambda calculus rather than a graph rewrite system. Second, our treatment of curried functions is completely different and makes the type system much simpler; in particular, there is no need for subtyping anymore. Finally, our system supports arbitrary rank types, and is much more careful about associating constraints with types.

The system as presented in this paper deals with the core lambda calculus only; however, extensions to deal with algebraic data types and recursive definitions are straightforward. For recursive definitions $\mu \cdot e$, the type of e is corrected to be non-unique (this is the same approach as taken in [6] for `letrec` expressions). The main principle in dealing with algebraic data types is that if a unique object is extracted from an enclosing container, the enclosing container must in turn be unique (this is a slightly more permissive definition than the one used in Clean).

We need to define a semantics for our small core language and show a number of standard properties of the type system with respect to this semantics (in particular, subject reduction). Also, we would like to prove that our system has principal types. Given an appropriate semantics with an explicit representation of sharing (for example, Launchbury's natural semantics for lazy evaluation [8], or perhaps a graph rewriting semantics), we should also prove that our type system does in fact guarantee that there is never more than one reference to an object with a unique type.

The inference algorithm described briefly in Sect. 6 is based on algorithm $\mathcal{W}$ and inherits its associated problems, in particular unhelpful error messages. We are planning to investigate the feasibility of other approaches; the constraint based algorithm proposed by Heeren looks promising [9].

The formalisation of the constraint language in this paper is not as precise as it could be, but a more precise definition is difficult to give. Moreover, the constraints considerably complicate the type system and the types assigned to terms. We are currently investigating whether it is possible to remove the constraints altogether by replacing the inequalities in the constraints by equalities. This will make the type system more restrictive, but will also make the type system much simpler. It remains to be seen whether this trade-off between simplicity and generality is desirable.

In the explanation of the rule for abstractions ABS in Sect. 3.4, it is mentioned that our method of constraining ν_a is conservative. For example, the constraint $u_{a'} \leq u$ in

$$\lambda x.\lambda y.y^{\odot} :: (a^u, c_1) \xrightarrow[u_a]{u_f} (b^v, c_2) \xrightarrow[u_{a'}]{u_{f'}} b^v, [c_2, u_{a'} \leq u]$$

is not actually necessary since x is not referenced in $\lambda y \cdot x$. It may be possible to relax the rules to be less conservative. That would only affect how ν_a is established; it would not change the type language.

Finally, the original motivation for wanting to extend Clean's uniqueness system to arbitrary rank is the fact that generic programming [10] frequently generates higher rank types. We plan to extend our prototype implementation of the system with support for generics, with the ultimate goal of proving that if a function defined generically is type correct (with respect to some "generic" uniqueness type system), then the functions derived from the generic function will also be type correct. This will also give us some experience with the type system, which may give us more insights into whether the extra power that our uniqueness system gives over Clean's system (see Sect. 7) is useful in practice.

Acknowledgements

We wish to thank Bastiaan Heeren, Dervla O'Keeffe, John Gilbert, Wendy Verbruggen, and Sjaak Smetsers for their comments on various drafts of this paper.

References

1. Barendsen, E., Smetsers, S.: Conventional and uniqueness typing in graph rewrite systems. Technical Report CSI-R9328, University of Nijmegen (1993)
2. Odersky, M., Läufer, K.: Putting type annotations to work. In: POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM Press (1996) 54–67
3. Peyton Jones, S., Shields, M.: Practical type inference for arbitrary rank types. Under consideration for publication in J. Functional Programming (2004)
4. 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
5. Huth, M., Ryan, M.: Logic in Computer Science: Modelling and Reasoning about Systems. Cambridge University Press, New York, NY, USA (2004)
6. Barendsen, E., Smetsers, S.: Uniqueness typing for functional languages with graph rewriting semantics. Mathematical Structures in Computer Science **6** (1996) 579–612
7. Wadler, P.: A taste of linear logic. In: MFCS '93: Proceedings of the 18th International Symposium on Mathematical Foundations of Computer Science, London, UK, Springer-Verlag (1993) 185–210
8. Launchbury, J.: A natural semantics for lazy evaluation. In: POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM Press (1993) 144–154
9. Heeren, B., Hage, J., Swierstra, S.D.: Generalizing Hindley-Milner type inference algorithms. Technical Report UU-CS-2002-031, Institute of Information and Computing Science, University Utrecht, Netherlands (2002)
10. Alimarine, A., Plasmeijer, M.J.: A generic programming extension for Clean. In: IFL '02: Selected Papers from the 13th International Workshop on Implementation of Functional Languages, London, UK, Springer-Verlag (2002) 168–185