Uniqueness Typing Simplified

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 present a uniqueness type system that is simpler than both Clean's uniqueness system and a system we proposed previously. The new type system is straightforward to implement and add to existing compilers, and can easily be extended with advanced features such as higher rank types and impredicativity. We describe our implementation in *Morrow*, an experimental functional language with both these features. Finally, we prove soundness of the core type system with respect to the call-by-need lambda calculus.

1 Introduction to Uniqueness Typing

An important property of pure functional programming languages is *referential transparency*: the same expression used twice must have the same value twice. This makes equational reasoning possible and aids program analysis, but most languages do not have this property. For example, in the following C fragment,

```
int f(FILE* file) {
  int a = fgetc(file); // Read a character from 'file'
  int b = fgetc(file);
  return a + b;
}
```

it is understood that `a` and `b` can have different values, even though we are applying the same function (`fgetc`) to the same input (`file`). Although the input is syntactically identical, the structure denoted by `file` is modified by each call to `fgetc` (the file pointer is advanced)—`fgetc` has a side effect.

In this example there would be no problem with referential transparency *if there was only a single reference to* `file`. A side effect on a variable (`file`) is okay as long as that variable is never used again: it is okay for a function to modify its input if the input is not *shared*. Referential transparency then trivially holds because the same expression never occurs more than once.

Uniqueness typing takes advantage of this observation to add side effects to a functional language without sacrificing referential transparency. The same function `f` implemented in a functional language using uniqueness typing gives

```
f file0 = let (a, file1) = fgetc file0
              (b, file2) = fgetc file1
          in (a + b, file2)
```

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

Rather than just returning the read character, `fgetc` returns a pair consisting of the read character *and a new file*, `file1`. Although `file0` and `file1` point to the same file on disk, they are conceptually *and* syntactically different, and thus it is clear that a and b may have different values. The uniqueness type system guarantees that `fgetc` is never applied to an argument which is used again (shared). For example, the type checker would reject

```
f file0 = let (a, file1) = fgetc file0
              (b, file2) = fgetc file0
          in (a + b, file0)
```

Sharing information is recorded as an attribute on the type of a term. This attribute is either $\bullet$ for unique (guaranteed not to be shared) or $\times$ for non-unique (may or may not be shared). For instance, $\texttt{File}^{\bullet}$ is the type of files that are guaranteed not to be shared, and the type of `fgetc` might be

$$\texttt{fgetc} :: \texttt{File}^{\bullet} \xrightarrow{\times} (\texttt{Char}^{\times}, \texttt{File}^{u})^{v}$$

The attribute on the arrow means that the function `fgetc` *itself* is non-unique (Sect. 4.2). The uniqueness variable u on the result means that it is up to the programmer to decide if they want to treat it as unique or shared (Sect. 6).

2 Contributions of This Paper

The type system we present in this paper is based on that of the programming language *Clean* [1, 2]. However, Clean's type system has a number of drawbacks.

- Types and attributes are regarded as two different entities, which limits expressiveness and impedes adding uniqueness typing to existing compilers.
- Types often involve implications between uniqueness attributes. For example, the function `const` has type

  $$\texttt{const} :: t^{u} \xrightarrow{\times} s^{v} \xrightarrow{w} t^{u}, [w \leq u]$$
  ```
  const x y = x
  ```

  The constraint $[w \leq u]$ denotes that if u is unique, then w must be unique (u implies w).[3] The need for this constraint will be explained in Sect. 4.2, but the presence of these constraints complicates the work of the type checker (the heart of the typechecker is a unification algorithm, and unification cannot deal with inequalities) and makes extending the type system to support modern features such as arbitrary rank types difficult.
- Clean distinguishes between non-unique terms, unique terms (which are unique now but may become non-unique later), and *necessarily unique* terms (which must remain unique forever). Moreover, Clean's type system has a subtyping relation between unique and non-unique terms. Both these features make the type system unnecessarily complicated.

[3] 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.

In this paper, we make the following contributions.

- Section 3 shows that we can regard uniqueness attributes as type constructors of a special kind. This increases the expressive power of the type system and simplifies the presentation and implementation of uniqueness typing.
- Section 4 presents the type system proper and shows how to avoid inequality constraints by allowing arbitrary boolean expressions as uniqueness attributes. This facilitates extending the type system with advanced features and enables the use of unification to solve relations between attributes.
- Section 6 shows how to avoid subtyping. We argued a similar point in a previous paper [3] but unfortunately the approach in that paper requires a second uniqueness attribute on the function arrow, offsetting the advantage of removing subtyping. Our new approach does not have this disadvantage.
- Section 7 describes our implementation in *Morrow*. Morrow supports higher rank types and impredicativity, but adding support for uniqueness typing to Morrow required only a few changes to the compiler. This provides strong evidence for our claim that retrofitting uniqueness typing to an existing compiler, and extending uniqueness typing with advanced features, becomes straightforward using the techniques in this paper. As far as the authors are aware, this is also the first substructural type system to have these features.
- Finally, we prove soundness of our type system in Sect. 8 with respect to the call-by-need lambda calculus [4].

3 Attributes Are Types

In this section, we show that we can regard types and attributes as one syntactic category. This simplifies both the presentation and implementation of a uniqueness type system and increases the expressive power of the type system.

If we regard types and attributes as distinct, we need type variables and attribute variables, and we need quantification ($\forall$) over type variables and attribute variables. In addition, the status of arguments to algebraic datatypes (such as `List a`) is unclear: are they types, attributes, or types *with* an attribute?

These issues are clarified when we regard types and attributes as a single syntactic category. Thus `Int` and `Bool` are types, and so are $\bullet$ (unique) and $\times$ (non-unique). We regard $\texttt{Int}^{\times}$ as syntactic sugar for the application of a special type constructor `Attr` to two arguments, `Int` and $\times$. There are no values of type $\times$, nor are there values of type `Int`, because `Int` is lacking a uniqueness attribute (there are however values of type $\texttt{Int}^{\times}$).

Types that do not classify values are not a new concept. For example, they arise in Haskell as *type constructors* such as the list type constructor (`[]`). We can make precise which types do and do not classify values by introducing a kind system [5]. Kinds can be regarded as the "types of types". By definition, the kind of types that classify values is denoted by $*$. In Haskell, we have `Int :: *`, `Bool :: *`, but `[] :: * → *`. The idea of letting the language of vanilla types and additional properties coincide is not new either (e.g., [6, 7]), but as far as the authors are aware it is new in the context of substructural type systems.

Kind language

κ	::=		kind
		$\mathcal{T}$	base type
		$\mathcal{U}$	uniqueness attribute
		$*$	base type together with a uniqueness attribute
		$\kappa_1 \to \kappa_2$	type constructors

Type constants

`Int`, `Bool`	::	$\mathcal{T}$	base type
$\to$	::	$* \to * \to \mathcal{T}$	function space
$\bullet$, $\times$	::	$\mathcal{U}$	unique, non-unique
$\vee$, $\wedge$	::	$\mathcal{U} \to \mathcal{U} \to \mathcal{U}$	disjunction, conjunction
$\neg$	::	$\mathcal{U} \to \mathcal{U}$	negation
`Attr`	::	$\mathcal{T} \to \mathcal{U} \to *$	combine a base type and attribute

Syntactic conventions

t^u	$\equiv$	`Attr` $t\ u$
$a \xrightarrow{u} b$	$\equiv$	`Attr` $(a \to b)\ u$

Fig. 1. The kind language and some type constructors with their kinds

Since we do not regard `Int` as a type classifying values, its kind cannot be $*$ in our type system. Instead, we introduce two new kinds, $\mathcal{T}$ and $\mathcal{U}$, classifying "base types" and uniqueness attributes. Since `Attr` combines a base type and an attribute into a type of kind $*$, its kind is $\mathcal{T} \to \mathcal{U} \to *$. The kind language and some type constructors along with their kinds are listed in Fig. 1. At this point it is useful to introduce the following convention.

> **(Syntactic convention.)** Type variables[4] of kind $\mathcal{T}$ and $\mathcal{U}$ will be denoted by t, s and u, v. Type variables of kind $*$ will be denoted by a, b.

One advantage of treating attributes as types is that we can use type variables to range over base types, uniqueness attributes or types with an attribute, simply by varying the kind of the type variable. This gives more expressive power when defining algebraic data types. For example, we can define:

```
newtype X a = MkX a
newtype Y t = MkY t^×
newtype Z u = MkZ Int^u
```

The type of a constructor argument must have kind $*$; hence, the first datatype is parameterized by an attributed type (a type of kind $*$), the second by a base type (a type of kind $\mathcal{T}$), and the third by a uniqueness attribute (a type of kind $\mathcal{U}$). The kinds of `X`, `Y` and `Z` are therefore $* \to \mathcal{T}$, $\mathcal{T} \to \mathcal{T}$ and $\mathcal{U} \to \mathcal{T}$, respectively. The codomain is $\mathcal{T}$ in all cases, since `X Int`$^\times$ still lacks an attribute; `(X Int`$^\times$`)`$^\bullet$ on the other hand is a unique `X` containing a non-unique `Int`. So, assuming $(5 :: \mathrm{Int}^\times)$, we have $(\texttt{MkX}\ 5 :: \texttt{X}^u\ \texttt{Int}^\times)$, $(\texttt{MkY}\ 5 :: \texttt{Y}^u\ \texttt{Int})$ and $(\texttt{MkZ}\ 5 :: \texttt{Z}^u\ \times)$.

[4] Strictly speaking, these are meta variables, not object language type variables. Our core language does not include universal quantification.

$e ::=$		expression	$\tau_k ::=$		type
	$x^{\odot}$	variable (used once)		c_k	constant
	$x^{\otimes}$	variable (used more than once)		$\tau_{(k' \to k)}\ \tau_{k'}$	type application
	$\lambda x \cdot e$	abstraction			
	$e\,e$	application			

Fig. 2. Expression and type language for the core system

In Clean, we can only define the first of these three datatypes, so we have gained expressive power. What is more, although we have used syntactic conventions to give a visual clue about the kinds of the type variables, the kinds of these types can automatically be inferred by the kind checker, so the expressive power comes at no cost to the programmer.

There are two possible variations to the kind system we propose. We could treat $\texttt{Int}^{\times}$ as the application of $(\texttt{Int} :: \mathcal{U} \to *)$ to $(\times :: \mathcal{U})$, or as the (postfix) application of $(\times :: \mathcal{T} \to *)$ to $(\texttt{Int} :: \mathcal{T})$, avoiding the need for `Attr`. We prefer distinguishing between $\mathcal{T}, \mathcal{U}$ and $*$, but if the reader feels otherwise they should feel free to read $\mathcal{T}$ as syntactic sugar for $(\mathcal{U} \to *)$, or $\mathcal{U}$ as syntactic sugar for $(\mathcal{T} \to *)$. In all three variations only types of kind $*$ are inhabited, as usual.

4 Removing Constraints

In this section we show that by allowing arbitrary boolean expressions[5] as uniqueness attributes (reading "true" for unique and "false" for non-unique) we can recode implications between uniqueness attributes as equalities. This makes the type system so similar to the classical Hindley/Milner type system that standard type inference algorithms can be applied and modern extensions such as arbitrary rank types can be incorporated without much difficulty.

The expression language and type language are defined in Fig. 2 (types have been indexed by their kind k). Both are almost entirely standard, except that we assume that a sharing analysis has annotated variable uses with $\odot$ or $\otimes$. A variable x marked as $x^{\odot}$ is used only once within its scope; a variable marked as $x^{\otimes}$ is used more than once. The typing rules are listed in Fig. 3. The typing relation takes the form

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

which reads as "in environment Γ, expression e has type τ; the attributes on the types of the free variables in e are fv". Both Γ and fv are mappings from term variables to types; the only difference is that Γ maps variables to types of kind $*$ and fv maps variables to types of kind $\mathcal{U}$ (in other words, to uniqueness attributes). The typing rule for abstraction uses fv to determine whether a function needs to be unique (this is discussed in more detail in Sect. 4.2).

[5] Although the typing rules only use disjunctions between uniqueness attributes, more complicated expressions can be introduced when unifying two boolean expressions.

$$\frac{}{\Gamma, x : t^u \vdash x^{\odot} : t^u|_{x:u}}\ \textsc{Var}^{\odot} \qquad \frac{}{\Gamma, x : t^{\times} \vdash x^{\otimes} : t^{\times}|_{x:\times}}\ \textsc{Var}^{\otimes}$$

$$\frac{\Gamma, x : a \vdash e : b|_{fv} \qquad fv' = \triangleright_x\, fv}{\Gamma \vdash \lambda x \cdot e : a \xrightarrow{\bigvee fv'} b|_{fv'}}\ \textsc{Abs}$$

$$\frac{\Gamma \vdash e_1 : a \xrightarrow{u} b|_{fv_1} \quad \Gamma \vdash e_2 : a|_{fv_2}}{\Gamma \vdash e_1\, e_2 : b|_{fv_1 \cup fv_2}}\ \textsc{App}$$

Fig. 3. Typing rules for the core lambda calculus

The rules are similar to the Hindley/Milner rules, except that they maintain some extra information about uniqueness. The underlying base system is unchanged, so that uniqueness typing can be seen as an "add-on".

4.1 Variables

We need to distinguish variables that are used once in their scope and variables that are used multiple times. The rule for variables that are used only once ($\textsc{Var}^{\odot}$) is identical to the normal Hindley/Milder rule, and we simply look up the type of the variable in the environment. Note that even when a variable is used only once, that does not automatically make its type unique. For example, there is only one use of `x` in the identity function:

```
id x = x⊙
```

but when a shared term is passed to `id`, it will still be shared when it is returned from `id`. On the other hand, if a variable is used more than once (rule $\textsc{Var}^{\otimes}$), its type must be non-unique (shared).

4.2 Partial Application

Dealing correctly with partial application is probably the most subtle aspect of uniqueness typing. We will demonstrate the problem using a simple example. Temporarily ignoring the attributes on arrows, the type of `dup` is

```
dup :: t× → (t×, t×)u
dup x = (x⊗, x⊗)
```

Since `dup` duplicates its argument, it only accepts non-unique arguments. The type checker can easily recognize that `dup` duplicates x because there is more than one use of x in the function body, which is therefore marked as $\otimes$. However, what if we rewrite `dup` as

```
dup' x = (\f -> (f⊗ ⊥, f⊗ ⊥)) (const x⊙)
```

Now there is only one reference to x, which is therefore marked as $\odot$. Still ignoring the attributes on arrows, the function `const` is defined as

```
const :: tu → sv → tu
const x y = x
```

It would therefore seem that the type of `dup'` is

$$\texttt{dup'} \;::\; t^u \to (t^u, t^u)^v$$

But that cannot be correct, because this type of `dup'` tells us that if we pass a single unique t to `dup'`, it will return a pair of two unique ts. However, the full type of `const` in our type system is

$$\texttt{const} \;::\; t^u \xrightarrow{\times} s^v \xrightarrow{u} t^u$$

If you pass in a unique t, you get a *unique* function from s to t: a function that can only be used once. Conversely, *if* you use a partial application of `const` more than once, the argument to `const` must be non-unique. The type of `dup'` is therefore

$$\texttt{dup'} \;::\; t^\times \xrightarrow{\times} (t^\times, t^\times)^u$$

Reassuringly, this is the same type as the type of `dup`. In general, a function must be unique (and can be applied only once) if it has any unique elements in its closure (the environment that binds the free variables in the function body).

4.3 Abstraction and Application

The rule for abstractions is the same as the Hindley/Milner rule, except that we must determine the value of the attribute on the arrow. As discussed in Sect. 4.2, a function must be unique if it has any unique elements in its closure. The closure of a function $\lambda x \cdot e$ consists of the free variables in the body e of the function, minus x. The attributes on the free variables in the body of the function are recorded in fv; using $fv' = \triangleright_x fv$ (domain subtraction) to denote fv with x removed from its domain, we use the disjunction $\bigvee fv'$ of all the attributes in the range of fv' as the uniqueness attribute on the arrow (recall that we treat uniqueness attributes as boolean expressions).

The rule for application is the normal one, except that we collect the free variables. The attribute on the arrow is ignored (we can apply both unique and shared functions).

4.4 Encoding Constraints

In general, we can always recode a type of the form

$$\ldots \square^u \quad \ldots \square^v \ldots, [u \le v]$$

using a disjunction

$$\ldots \square^{u \vee v} \ldots \square^v \ldots$$

This faithfully models the implication: when v is unique, $u \vee v$ reduces to unique, but when v is non-unique, $u \vee v$ reduces to u. For example, in Clean the function `fst` that extracts the first element of a pair has the type

$$\texttt{fst} \;::\; (t^u, s^v)^w \to t^u, [w \le u]$$

```
fst (x, y) = x
```

which we can recode as

$$\texttt{fst} \;::\; (t^u, s^v)^{w \vee u} \to t^u$$

However, in many cases we can do slightly better. For example, suppose the typing rule for pairs is

$$\frac{\Gamma \vdash e_1 : a|_{fv} \qquad \Gamma \vdash e_2 : b|_{fv}}{\Gamma \vdash (e_1, e_2) : (a, b)^u|_{fv}} \quad \textsc{Pair}$$

then for every derivation of $e :: (a, b)^\bullet$, there is also a derivation of $e :: (a, b)^\times$ (because the typing rule leaves the attribute on the pair free). That means that we can simplify the type of `fst` to

$$\texttt{fst'} \;::\; (t^u, s^v)^u \to t^u$$

The only pairs accepted by `fst` but rejected by `fst'` are unique pairs, but since the type checker will never infer a pair to be unique (but always either non-unique or polymorphic in its uniqueness), that situation will never arise. We took advantage of the same principle in the rule for abstraction, where we recoded a type

$$\ldots \xrightarrow{u} \ldots, [u \leq v, u \leq w, \ldots]$$

as

$$\ldots \xrightarrow{v \vee w \vee \ldots} \ldots$$

This will force some functions to be non-unique which would otherwise be polymorphic in their uniqueness, but that cannot cause any type errors: the rule for function application ignores the uniqueness attribute on the function, and non-unique functions can be used multiple times.

5 Boolean Unification

One advantage of removing constraints from the type language is that standard inference algorithms (such as algorithm $\mathcal{W}$ [8]) can be applied without any modifications. The inference algorithm will depend on a unification algorithm, which must be modified to use boolean unification when unifying two terms of kind $\mathcal{U}$. Suppose we have two terms g and h

$$g :: t^\bullet \xrightarrow{\times} \ldots \qquad h :: t^{u \vee v}$$

Should the application $g\,h$ be allowed? If so, we must be able to unify $u \vee v \simeq \bullet$. This equation has many solutions, such as $[u \mapsto \bullet, v \mapsto v]$, $[u \mapsto u, v \mapsto \bullet]$, or $[u \mapsto \bullet, v \mapsto \bullet]$. (Recall that attributes are boolean expressions.) However, none of these solutions is most general, and it is not obvious that the equation $u \vee v \simeq \bullet$ even has a most general unifier, which means we would lose principal types. Fortunately, unification in a boolean algebra is unitary [9]. In other words, if a boolean equation has a solution, it has a most general solution. In the example, one most general solution is $[u \mapsto u, v \mapsto v \vee \neg u]$.

```
unify₀ :: BooleanAlgebra a => a -> [Var] -> (Subst a, a)
unify₀ t [] = ([], t)
unify₀ t (x : xs) = (s_t ∪ s_e, cc)
  where s_t = [x ↦ s_e t₀ ∨ (x ∧ s_e (¬t₁))]
        (s_e, cc) = unify₀ (t₀ ∧ t₁) xs
        t₀ = [x ↦ 0] t
        t₁ = [x ↦ 1] t
```

Fig. 4. Boolean unification (unify $t \simeq 0$)

There are two well-known algorithms for unification in a boolean algebra, known as Löwenheim's formula and successive variable elimination [9, 10]. For our core system either algorithm will work, but when arbitrary rank types are introduced and we need to use skolemization [11], only the second is practical.[6] Temporarily using the more common 0 for false (not unique) and 1 for true (unique), to unify two terms p and q it suffices to unify $t = (p \wedge \neg q) \vee (\neg p \wedge q) = 0$. This is implemented by `unify`$_0$, shown in Fig. 4, which takes 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 On Subtyping

In this section we compare our approach to subtyping with that of Clean [2] and to that of our previous paper on the topic [3]. Consider again the function `dup`:

```
dup :: t^× -×-> (t^×, t^×)^u
dup x = (x, x)
```

In Clean `dup` has the same type, but that type is interpreted differently. Clean's type system uses a subtyping relation: a unique type is considered a subtype of a non-unique type. That is, we can pass in something that is unique (such as a unique `Array`) to a function that is expecting a non-unique type (such as `dup`).

The fact that a unique array can become non-unique is an important feature of a uniqueness type system. A non-unique array can no longer be updated, but can still be read from. However, adding subtyping to a type system leads to considerable additional complexity, especially when considering a contravariant/covariant system with support for algebraic data types (such as Clean's). It becomes simpler when considering an invariant subtyping relation, but we feel that subtyping is not necessary at all.

[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 easy, but it is more difficult in the presence of skolem constants. For example, assuming that u_R and v_R are skolem constants, and w is a uniqueness variable, the equation $u_R \vee v_R \simeq w$ has a trivial solution $[w \mapsto u_R \vee v_R]$, but we can no longer guess this solution by instantiating all variables to either true or false.

In our previous paper, we argued that the type of `dup` should be

$$\texttt{dup} \;::\; t^u \xrightarrow[\times]{u_f} (t^\times, t^\times)^v$$

The (free) uniqueness variable on the t in the domain of the function indicates that we can pass unique or non-unique terms to `dup`. Since it is always possible to use a uniqueness variable in lieu of a non-unique attribute, an explicit subtyping relation is not necessary.

But there is a catch. As we saw in Sect. 4.2, functions with unique elements in their closure must be unique, and must *remain* unique: they should only be applied once. In Clean, this is accomplished by regarding unique functions as *necessarily unique*, and the subtyping is adjusted to deal with this third notion of uniqueness: a necessarily unique type is *not* a subtype of a non-unique type. Hence, we cannot pass functions with unique elements in their closure to `dup`.

Unfortunately, when `dup` gets the type from our previous paper it *can* be used to duplicate functions with unique elements in their closure. Therefore we introduced a second attribute on the function arrow, indicating whether the function had any unique elements in its closure. The typing rule for application enforced that functions with unique elements in their closure (second attribute) were unique (first attribute). That means that functions with unique elements in their closure can be duplicated, but once duplicated can no longer be applied.

This removed the need for subtyping, but that advantage was offset by the additional complexity introduced by the second uniqueness attribute on arrows: the additional attribute made types more difficult to read (especially in the case of higher order functions).

An important contribution of the current paper is the observation that this additional complexity can be avoided if we are careful when assigning types to library functions. For example, a function that returns a new empty array should get the type

$$\texttt{newArray} \;::\; \texttt{Int} \xrightarrow{\times} \texttt{Array}^u$$

rather than

$$\texttt{newArray} \;::\; \texttt{Int} \xrightarrow{\times} \texttt{Array}^\bullet$$

Similarly, the function that clears all elements of an array should get the type

$$\texttt{resetArray} \;::\; \texttt{Array}^\bullet \xrightarrow{\times} \texttt{Array}^u$$

rather than

$$\texttt{resetArray} \;::\; \texttt{Array}^\bullet \xrightarrow{\times} \texttt{Array}^\bullet$$

An `Array` that is polymorphic in its uniqueness can be passed to `resetArray` as easily as it can be passed to `dup` (of course, a shared array still cannot be passed to `resetArray`). If we are careful never to *return* a unique array from a function, we will always be able to share arrays. We still do not have an explicit subtyping relation but we get the same functionality: the subtyping is encoded in the type of `Array`, rather than in the type of `dup`.

Not all functions should be so modified. For example, many functions with side effects in Clean have a type such as

$$\texttt{fun} \;::\; \cdots \to (\texttt{World}^{\bullet} \to \texttt{World}^{\bullet})$$

where the `World` is a token object representing the world state. It never makes sense to duplicate the world, which can be enforced by returning a unique `World` (rather than a `World` which is polymorphic in its uniqueness).

It may seem that a disadvantage of our approach is that we can no longer take advantage of more advanced sharing analyses. For example, given

$$\begin{aligned} &\texttt{isEmpty} && :: \texttt{Array}^{u} \xrightarrow{\times} \texttt{Bool}^{\times} \\ &\texttt{shrink}\,,\ \texttt{grow} && :: \texttt{Array}^{\bullet} \xrightarrow{\times} \texttt{Array}^{u} \end{aligned}$$

sharing analysis has been applied correctly to the following definition [2]:

$$\texttt{f arr = if isEmpty arr}^{\otimes}\ \texttt{then shrink arr}^{\odot}\ \texttt{else grow arr}^{\odot}$$

Even though there are three uses of `arr` within `f`, only one of the two branches of the `if`-statement will be executed. Moreover, the condition is guaranteed to be evaluated before either of the branches, and the shared ($\otimes$) annotation on `arr` means that the array will not be modified when the condition is evaluated.

However, this example uses `arr` at two different types: $\texttt{Array}^{\times}$ within the condition and $\texttt{Array}^{\bullet}$ within both branches. This works in Clean because $\texttt{Array}^{\bullet}$ is a subtype of $\texttt{Array}^{\times}$. In our previous proposal [3], this works because a unique term can always be considered as a non-unique term. In our new proposal however, this program would be rejected (since $\texttt{Array}^{\bullet}$ does not unify with $\texttt{Array}^{\times}$).

However, we can take advantage of the fact that we have embedded our core system in an advanced type system that supports first class polymorphism (Sect. 7). We want to use a polymorphic value ($\texttt{arr} :: \forall u.\texttt{File}^{u}$) at two different types within a function: the classic example of a higher rank type [11]. Our example above typechecks if we provide the following type annotation:

$$\texttt{f} \;::\; \forall v.\,(\forall u.\,\texttt{Array}^{u}) \xrightarrow{\times} \texttt{Array}^{v}$$

The function `f` now *demands* that the array that is passed in is polymorphic in its uniqueness. That is reasonable when we consider that we are using the array at two different types in the body. Moreover, since we regard all unique objects as necessarily unique, it is also reasonable that we cannot pass in a truly unique array to `f`.

Of course there is a trade-off here between simplicity (and ease of understanding) of the type system on the one hand and usability on the other. Since the user must provide a type annotation in order for the definition of `f` to typecheck, the type system has arguably become more difficult to use. However, this case is rare enough that the additional burden on the programmer is small, and a case can be made that it is useful to require a type annotation as it is non-obvious why the function definition is accepted.

7 Implementation in Morrow

We have integrated our type system in *Morrow*, an experimental functional language developed by Daan Leijen.[7] Morrow's type system is HMF [12], which is a Hindley/Milner-like type system that supports first class polymorphism (higher rank types and impredicativity). As such, it is an alternative to both Boxy Types [13] and MLF [14]. However, unlike boxy types, it is presented as a small logical system which makes it easier to understand, and at the same time it is much simpler than MLF. Although HMF is quite a good fit with our type system, we have also experimented with integrating it into other type systems. For example, we we have a prototype implementation of a variant on the type system of this paper that uses the arbitrary rank type system from [11].

As it turns out, the implementation of our type system in Morrow is agreeably straightforward. This provides strong evidence for our claim that adding uniqueness typing to an existing compiler, and more importantly, extending uniqueness typing with advanced features such as higher rank types and impredicativity, poses little difficulty when using the techniques from this paper.

We outline the most important changes we had to make to Morrow:

- We modified the kind checker to do kind inference for our new kind system (mostly a matter of changing the kinds of type constants)
- We implemented sharing analysis, annotating variables with information on how often they are used within their scope (once or more than once)
- We modified the rules for variables and abstraction, so that shared variables must be non-unique, and abstractions become unique when they have unique elements in their closure. To be able to do the latter, all the typing rules had to be adapted to return the fv structure from Sect. 4. Variables that are used at a polymorphic uniqueness (a type of the form $\forall u.t^u$ for some t) must be treated as if they were unique for the purposes of fv.
- Let bindings had to be adapted to remove the variables bound from fv. Moreover, the type of every binding in a recursive binding group must be non-unique (as is standard in a uniqueness type system [2]).
- Most of the work was in modifying the types of the built-in functions and the kinds of the built-in types, and adding the appropriate type constants (such as `Attr`) and kind constants ($\mathcal{T}, \mathcal{U}$). However, all of these changes were local and did not affect the rest of the type checker.
- Unification had be adapted to do boolean unification, as explained in Sect. 5. In addition, it is necessary to simplify boolean expressions, so that for example $t^{u \vee \times}$ is simplified to t^u. This is important because if no simplification is used the boolean expressions can quickly get complicated. Fortunately, we can use an independent module for boolean unification and simplification. When unifying $a \simeq b$, it suffices to check the kinds of a and b, and if they are $\mathcal{U}$, to call the boolean unification module. Therefore, boolean unification does not in any way complicate the unification algorithm of the type checker.

[7] Unfortunately we cannot currently make the source available due to licensing issues.

– Morrow uses System F (with pattern matching) as its typed internal language. Although the "attributes are types" approach of Sect. 3 means that the internal language does not need to change, Morrow also includes a System F type checker to ensure that the various phases of the compiler generate valid code. This type checker had to be adapted in a similar way to the main type checker.

The majority of these changes were local (did not require any significant refactoring of the compiler), and none of the changes were complicated. The fact that we can treat both vanilla types and uniqueness attributes as types (of different kinds) really helped: modifying the kind checker was straightforward, we got the additional expressive power described in Sect. 3 virtually for free, we did not have to introduce an additional universal quantifier for uniqueness attributes (and thus avoided having to modify operations on types such as capture avoiding substitution or pretty-printing), etc.

8 Soundness

To prove soundness, we use a slightly modified (but equivalent) set of typing rules.[8] Rather than giving different typing rules for variables marked as used once or used more than once, we do not mark variables at all but enforce that unique variables are used at most once by splitting the environment into two in rule App. Non-unique variables can still be used more than once because the context splitting operation collapses multiple assumptions about non-unique variables (rule Split$^{\times}$). This presentation of the type system is known as a *substructural* presentation because some of the structural rules (in this case, contraction) do not hold. The presentation style we have used, using a context splitting operation, is based on that given in [15], where it is attributed to [16].

The soundness proof for a type system states that when a program is well-typed it will not "go wrong" when evaluated with respect to a given semantics. We are interested in a lazy semantics; often the call-by-name lambda calculus is used as an approximation to the lazy semantics, but it is not hard to see that we will not be able to prove soundness with respect to the call-by-name semantics. For example, consider

```
(\x. (x, x)) (f y)
```

In the call-by-name semantics, this term evaluates to

```
(f y, f y)
```

But when we allow for side effects, these two terms have a different meaning. In the first, we evaluate $f\,y$ once and then duplicate the result; in the second, we

[8] The syntax-directed presentation using sharing marks is easier to understand and more suitable for type inference. However, it is not usable for a soundness proof. Such a distinction between a syntax-directed and a logical presentation is not uncommon, and has been used before in the context of uniqueness typing [2].

Term language

$e ::= x \mid \lambda x \cdot e \mid e\, e$	term
$A ::= \lambda x \cdot e \mid \text{let } x = e \text{ in } A$	answer
$E ::= [] \mid E\, e \mid \text{let } x = e \text{ in } E \mid \text{let } x = E_0 \text{ in } E_1[x]$	evaluation context

Syntactic convention

$(\text{let } x = e_1 \text{ in } e_2) \equiv (\lambda x \cdot e_2)\, e_1$

Evaluation rules

$\mapsto$ is the smallest relation that contains Value, Commute, Assoc and is closed under the implication $M \mapsto N$ implies $E[M] \mapsto E[N]$.

(Value)	$\text{let } x = \lambda y \cdot e \text{ in } E[x]$	$\mapsto \{(\lambda y \cdot e)/x\}\, E[x]$
(Commute)	$(\text{let } x = e_1 \text{ in } A)\, e_2$	$\mapsto \text{let } x = e_1 \text{ in } A\, e_2$
(Assoc)	$\text{let } y = (\text{let } x = e \text{ in } A) \text{ in } E[y]$	$\mapsto \text{let } x = e \text{ in let } y = A \text{ in } E[y]$

Substructural typing rules

$$\frac{}{\Gamma, x : t^u \vdash x : t^u|_{x:u}}\ \textsc{Var}$$

$$\frac{\Gamma, x : a \vdash e : b|_{fv} \qquad fv' = \triangleright_x fv}{\Gamma \vdash \lambda x \cdot e : a \xrightarrow{\bigvee fv'} b|_{fv'}}\ \textsc{Abs}$$

$$\frac{\Gamma \vdash e_1 : a \xrightarrow{u} b|_{fv_1} \quad \Delta \vdash e_2 : a|_{fv_2}}{\Gamma \circ \Delta \vdash e_1\, e_2 : b|_{fv_1 \circ fv_2}}\ \textsc{App}$$

Context splitting

$$\frac{}{\emptyset = \emptyset \circ \emptyset}\ \textsc{Split}^{\emptyset} \qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x : t^{\times} = (\Gamma_1, x : t^{\times}) \circ (\Gamma_2, x : t^{\times})}\ \textsc{Split}^{\times}$$

$$\frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x : t^{\bullet} = (\Gamma_1, x : t^{\bullet}) \circ \Gamma_2}\ \textsc{Split}_1^{\bullet} \qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x : t^{\bullet} = \Gamma_1 \circ (\Gamma_2, x : t^{\bullet})}\ \textsc{Split}_2^{\bullet}$$

Fig. 5. Call-by-Need Semantics

evaluate $f\, y$ twice (and so have the potential side effect of f twice). Accordingly, the types of both terms in a uniqueness type system are also different. In the first, f may or may not be unique, and must have a non-unique result (because the result is duplicated). In the second, f cannot be unique (because it is applied twice) and may or may not return a unique result.

Traditionally [2] a graph rewriting semantics is used to prove soundness, but this complicates equational reasoning. Fortunately, it is possible to give an algebraic semantics for lazy evaluation. Launchbury's *natural semantics for lazy evaluation* [17] is well-known and concise, but is a big-step semantics which makes it less useful for a soundness proof. The call-by-need semantics by Maraist *et al.* [4] is slightly more involved, but is a small-step semantics and fits our needs perfectly. The semantics is shown in Fig. 5.

Unfortunately, due to space limitations we can only give a summary of the proof here. A full formal proof, written using the *Coq* proof assistant, can be found in a separate technical report [18].

Theorem 1 (Progress). *Suppose e is a closed, well-typed term ($\emptyset \vdash e : \tau|_{fv}$ for some τ and fv). Then either e is an answer or there exists some e' such that $e \mapsto e'$.*

Proof. The easiest way to prove progress is to prove a weaker property first: for every term e, e is an answer, there exists some e' such that $e \mapsto e'$, or $e = E[x]$ for some x. This weaker property can be proven by a complete structural induction on e; the proof is laborious but not difficult. To prove progress using the weak progress property, we just need to rule out the last possibility. However, if $e = E[x]$ for some x, and $\emptyset \vdash e : \tau|_{fv}$, then we must have $x \in \emptyset$, which is impossible. □

The proof of preservation is more involved and we can only give a brief outline here. The main lemma that we need to be able to prove preservation is the substitution lemma:

Lemma 1 (Substitution). *If $\Gamma, x : a \xrightarrow{\bigvee fv_2} b \vdash e_1 : \tau|_{fv_1, x:\bigvee fv_2}$, x is free in e_1, and $\Delta \vdash \lambda y \cdot e_2 : a \xrightarrow{\bigvee fv_2} b|_{fv_2}$, then $\Gamma \circ \Delta \vdash \{(\lambda y \cdot e_2)/x\}\, e_1 : \tau|_{fv_1 \circ fv_2}$.*

The proof is by induction on $\Gamma, x : a \xrightarrow{\bigvee fv_2} b \vdash e_1 : \tau|_{fv_1, x:\bigvee fv_2}$ and is not trivial. The essence of the proof is that if $(\lambda x \cdot e_1)(\lambda y \cdot e_2)$ is well-typed, then either x occurs once in e_1, in which case we can substitute $\lambda y \cdot e_2$ for x without difficulty, or x occurs more than once in e_1. In that case, x must have a non-unique type, which means that $\lambda y \cdot e_2$ must be non-unique, and therefore the function cannot have any unique elements in its closure—or equivalently, that e_2 be typed in an environment where every variable has a non-unique type. Since $\Delta = \Delta \circ \Delta$ if all assumptions in Δ are non-unique, this means that we can type the result term even when $\lambda y \cdot e_2$ is duplicated.

Armed with the substitution lemma, we can prove preservation:

Theorem 2 (Preservation). *If $\Gamma \vdash e : \tau|_{fv}$ and $e \mapsto e'$ then $\Gamma \vdash e' : \tau|_{fv}$.*

Proof. By induction on $e \mapsto e'$. The cases for Commute, Assoc, and the three closure rules (one for each of the non-trivial evaluation contexts) are reasonably straightforward. The case for Value relies on the substitution lemma. □

A full formalization of the calculus extended with (let-bound or first-class) polymorphism is future work.

9 Related Work

There is a large body of related work; we can only discuss the most relevant.

There are two recent papers on uniqueness typing: Harrington [19] presents a categorical semantics for a uniqueness type system like Clean's, and Hage *et al.* [20] present a generic type system that can be instantiated to support either sharing analysis or uniqueness typing.

In both systems all unique terms can be coerced to non-unique terms. As observed in Sect. 6 it is possible to allow this, but one must be careful with partially applied functions which may have unique elements in their closure.

In the type system from Hage *et al.*, functions with unique elements in their closure must be unique; however, these functions can then be coerced to be non-unique and can be applied an arbitrary many times; no special provision is made to prohibit this. Thus, it is possible to define a function `dup!` of type

$$\texttt{dup!} :: t^{\bullet} \to (t^{\bullet}, t^{\bullet})^{v}$$

```
dup! x = (\f -> (f ⊥, f ⊥)) (const x)
```

The authors suggest that the problem may be remedied by introducing an additional attribute on arrows, like we suggested in our previous paper (see also Sect. 6)—and they adopt this solution in a later paper [21]. It remains to be seen whether a similar solution to the one we propose in the current paper is possible for their system. The central thesis of their paper is a duality between uniqueness typing and sharing analysis, and it is not clear whether this duality is preserved when removing subtyping.

Harrington suggests a different solution to the problem of partial application. Two distinct sorts of functions are introduced: ones that can have unique elements in their closure (of type $a \multimap b$) and ones that cannot (of type $a \Rightarrow b$). Functions of type $a \Rightarrow b$ do not pose any problems and can safely be applied many times (and potentially return unique results).

Functions with unique elements in their closure can also be applied multiple times, but their result must be non-unique if they are applied more than once. While this means that it is no longer possible to define `dup!`, this approach is not sufficient to guarantee referential transparency. For example, consider a function `closeFile` which returns a boolean indicating whether the file was already closed:

$$\texttt{closeFile} :: \texttt{File}^{\bullet} \xrightarrow{\times} \texttt{Bool}^{\times}$$

In Harrington's system, the following program would be accepted

```
f file = (\g. g ⊥, g ⊥) (\x. closeFile file)
```

even though it is not referentially transparent (it would be rejected in our type system). It is accepted because the `closeFile` *always* returns a non-unique result, and hence the restriction that functions that are used more than once must return a non-unique result makes no difference (and hence is not enough to guarantee referential transparency). It may be difficult to modify Harrington's system to adopt a solution similar to the one we propose: subtyping between unique and non-unique terms is fundamental to Harrington's formalization.

Uniqueness typing is often compared to linear (or affine) logic [22]. Although both linear logic and uniqueness typing are substructural logics, there are important differences. In linear logic, variables of a non-linear type can be coerced

to a linear type (dereliction). Harrington phrases it well: in linear logic, "linear" means "will not be duplicated" whereas in uniqueness typing, "unique" means "has not been duplicated". According to Wadler: "Does this mean that linearity is useless for practical purposes? Not completely. Dereliction means that we cannot guarantee a priori that a variable of linear type has exactly one pointer to it. But if we know this by other means, then linearity guarantees that the pointer will not be duplicated or discarded" [22, Sect. 3].

However, some systems based on linear logic (such as [23]) are much closer to uniqueness typing than to linear logic, and these systems could benefit equally from the techniques presented in this paper (attributes as types, boolean expressions for attributes).

Finally, Guzmán's Single-Threaded Polymorphic Lambda Calculus [24] has similar goals to uniqueness typing, but is considerably more complicated. Much of this complexity comes from trying to support a "strict let" construct where unique (or "single-threaded") terms can be used multiple times at a non-unique (multiple-threaded) type. A detailed discussion of this problem is beyond the scope of this paper; see for example [25, Sect. 9.4] or [26].

10 Conclusions

By treating uniqueness attributes as types of a special kind $\mathcal{U}$, the presentation and implementation of a uniqueness type system is simplified, and we gain expressiveness in the definition of algebraic datatypes. We can recode attribute inequalities (implications between uniqueness variables) as equalities if we allow for arbitrary boolean expressions as uniqueness attributes. This makes type inference easier (unification cannot deal with inequalities, but *can* deal with equalities between boolean expressions). Finally, no explicit subtyping relation is necessary if we are careful when assigning types to library functions: we require that unique terms must never be shared, and make sure that functions never return unique terms (but rather terms that are polymorphic in their uniqueness).

Together these observations lead to an expressive yet simple uniqueness type system, which is sound with respect to the call-by-need lambda calculus. The system can easily be extended with advanced features such as higher rank types and impredicativity. We have integrated our type system in Morrow, an experimental programming language with an advanced type system. The implementation required only minor changes to the compiler, providing strong evidence for our claim that retrofitting our type system to existing compilers is straightforward.

Acknowledgements. We thank Daan Leijen, Paul Levy and Adam Megacz for various insightful discussions, and Arthur Charguéraud for his generous assistance with the formal proof in *Coq*, which uses the proof engineering technique devised by him and others [27].

References

1. Barendsen, E., Smetsers, S.: Conventional and uniqueness typing in graph rewrite systems. Technical Report CSI-R9328, University of Nijmegen (December 1993)
2. Barendsen, E., Smetsers, S.: Uniqueness typing for functional languages with graph rewriting semantics. Math. Struct. in Computer Science **6** (1996) 579–612
3. De Vries, E., Plasmeijer, R., Abrahamson, D.: Uniqueness typing redefined. In Horváth, Z., Zsók, V., Butterfield, A., eds.: IFL 2006. (2007)
4. Maraist, J., Odersky, M., Wadler, P.: The call-by-need lambda calculus. JFP **8**(3) (1998) 275–317
5. Jones, M.P.: A system of constructor classes: overloading and implicit higher-order polymorphism. In: FPCA '93. (1993) 52–61
6. Sheard, T.: Putting Curry-Howard to work. In: Haskell Workshop '05, ACM (2005) 74–85
7. Sulzmann, M., Chakravarty, M.M.T., Jones, S.P., Donnelly, K.: System F with type equality coercions. In: TLDI '07, ACM (2007) 53–66
8. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: POPL '82. (1982) 207–212
9. Baader, F., Niphow, T.: Term Rewriting and All That. Cambridge University Press (1998)
10. Brown, F.M.: Boolean Reasoning. Dover Publications, Inc. (2003)
11. Peyton Jones, S., Vytiniotis, D., Weirich, S., Shields, M.: Practical type inference for arbitrary-rank types. JFP **17**(1) (Jan 2007) 1–82
12. Leijen, D.: HMF: Simple type inference for first-class polymorphism. Technical Report MSR-TR-2007-118, Microsoft Research, Redmond
13. Vytiniotis, D., Weirich, S., Peyton Jones, S.: Boxy types: inference for higher-rank types and impredicativity. In: ICFP '06. (2006) 251–262
14. Botlan, D.L., Rémy, D.: ML^F: raising ML to the power of System F. In: ICFP '03. (2003) 27–38
15. Walker, D.: Substructural type systems. In Pierce, B., ed.: Advanced Topics in Types and Programming Languages. The MIT Press (2005)
16. Cervesato, I., Pfenning, F.: A linear logical framework. Inf. Comput. **179**(1) (2002) 19–75
17. Launchbury, J.: A natural semantics for lazy evaluation. In: POPL '93. (1993) 144–154
18. de Vries, E.: Uniqueness typing simplified—technical appendix. Technical Report TCD-CS-2008-19, Trinity College Dublin
19. Harrington, D.: Uniqueness logic. Theor. Comput. Sci. **354**(1) (2006) 24–41
20. Hage, J., Holdermans, S., Middelkoop, A.: A generic usage analysis with subeffect qualifiers. In: ICFP '07, ACM (2007) 235–246
21. Hage, J., Holdermans, S.: Heap recycling for lazy languages. In: PEPM '08, ACM (2008) 189–197
22. Wadler, P.: Is there a use for linear logic? In: PEPM '91. (1991) 255–273
23. Turner, D.N., Wadler, P., Mossin, C.: Once upon a type. In: FPCA '95. (1995) 1–11
24. Guzman, J., Hudak, P.: Single-threaded polymorphic lambda calculus. In: Logic in Computer Science '90. (June 1990) 333–343
25. Plasmeijer, R., van Eekelen, M.: Clean language report (version 2.1)
26. Odersky, M.: Observers for linear types. In: ESOP '92, Springer-Verlag (1992) 390–407
27. Aydemir, B., Charguéraud, A., Pierce, B.C., Pollack, R., Weirich, S.: Engineering formal metatheory. SIGPLAN Not. **43**(1) (2008) 3–15