Modelling Unique and Affine Typing using Polymorphism

Edsko de Vries

Abstract. Uniqueness typing and affine (or linear) typing are dual type systems. Uniqueness gives a *guarantee* that an term *has not* been shared, while affinity imposes a *restriction* that a term *may not* be shared. We show that we can unify both concepts through polymorphism.

1 Introduction

Side effects in modern pure functional languages such as Clean or Haskell are modelled as functions that transform the world. For instance, a function that reads a character from the keyboard might have type

$$\texttt{getChar} :: \texttt{World} \to (\texttt{World}, \texttt{Char})$$

The return type of `getChar` makes it clear that c_1 and c_2 can have different values in

$$\begin{aligned} \lambda world \cdot \text{let } (c_1, world') &= \texttt{getChar}\ world \\ (c_2, world'') &= \texttt{getChar}\ world' \\ \text{in } (c_1, c_2, world'') & \end{aligned}$$

They are read in different worlds, after all. Of course, this is a symbolic representation of the world only, which means we somehow need to outlaw programs such as

$$\begin{aligned} \lambda world \cdot \text{let } (c_1, world') &= \texttt{getChar}\ world \\ (c_2, world'') &= \texttt{getChar}\ world \\ \text{in } (c_1, c_2, world'') & \end{aligned} \tag{1}$$

One way to do this is to define an opaque wrapper type

$$\texttt{IO}\ a \mathrel{\hat{=}} \texttt{World} \to (\texttt{World}, a)$$

together with two operations

$$\begin{aligned} \texttt{return} &:: a \to \texttt{IO}\ a \\ \texttt{bind} &:: \texttt{IO}\ a \to (a \to \texttt{IO}\ b) \to \texttt{IO}\ b \end{aligned}$$

That is, define `IO` to be a *monad*. Since the plumbing of the `World` happens inside `bind` "reusing" the same world cannot happen. This is the approach taken in Haskell.

An alternative approach is to use a type system to outlaw programs such as (1). For instance, we can use Clean's *uniqueness typing* to give `getChar` the type

$$\texttt{getChar} :: \texttt{World}^{\bullet} \to (\texttt{World}^{\bullet}, \texttt{Char})$$

The annotation on $\texttt{World}^{\bullet}$ means that `getChar` requires a *unique*—or non-shared—reference to the world and in turn promises to return a unique reference.

An advantage of this approach over the use of monads is that it is more compositional. For example, we can easily define a function that modifies two arrays *in place*

$$\begin{aligned}&\texttt{modifyArrays} :: (\texttt{Array}^{\bullet}, \texttt{Array}^{\bullet}) \to (\texttt{Array}^{\bullet}, \texttt{Array}^{\bullet})\\&\texttt{modifyArrays} = \ldots\end{aligned}$$

without specifying in which order these two updates should happen (indeed, they could happen in parallel).

Uniqueness typing is a *substructural* logic. We will explain this in more detail in Sect. 2. Probably the most well-known substructural logic is affine (or linear) logic. Affine logic can be regarded as *dual* to uniqueness typing; we discuss it in more detail in Sect. 3. In Sect. 4 we observe that we can simplify and unify both type systems through a familiar typing construct: polymorphism. We show that there is a sound translation from unique and affine typing into the unified system, and argue that although the translation is not complete, the loss is outweighed by the benefits of unifying the two systems. Finally, we wrap in Sect. 6.

2 Uniqueness Typing

The type syntax that we will use throughout this paper is given by

$$\begin{array}{lll}\alpha ::= \bullet \mid \times & & \text{(type attribute)}\\ \tau ::= c \mid \sigma \underset{\alpha}{\to} \sigma' & & \text{(base type)}\\ \sigma ::= \tau^{\alpha} & & \text{(attributed type)}\\ c \in \texttt{Unit}, \texttt{Bool}, \texttt{Array}, \ldots & & \text{(constants)}\end{array}$$

where we will write $(\sigma_1 \underset{\alpha'}{\to} \sigma_2)^{\alpha}$ as $\sigma_1 \overset{\alpha}{\underset{\alpha'}{\to}} \sigma_2$, and we will occasionally follow Clean convention and use the absence of a type annotation to mean non-unique (i.e., we might write $\tau^{\times}$ as τ). The reason for the additional attribute on the function arrow will become clear in Sect. 2.3.

In the context of uniqueness typing the attribute "$\bullet$" is read as "unique" (guaranteed not shared), and the attribute "$\times$" is pronounced "non-unique" (possibly shared). The typing rules for uniqueness typing are shown in Figure 1.

2.1 Contraction

Typing environments (here and elsewhere in this paper) are *bags* of pairs of identifiers and types, *not sets*. That is, the typing environment $\{x : \sigma, x : \sigma\}$ with

Logical Rules

$$\frac{}{x : \sigma \vdash x : \sigma}\textsc{Var}$$

$$\frac{\Gamma, x : \sigma_1 \vdash e : \sigma_2}{\Gamma \vdash \lambda x \cdot e : \sigma_1 \xrightarrow[\sup \Gamma]{\alpha} \sigma_2}\textsc{Abs} \qquad \frac{\Gamma_1 \vdash e_1 : \sigma_1 \xrightarrow[\alpha']{\alpha} \sigma_2 \quad \Gamma_2 \vdash e_2 : \sigma_1 \quad \alpha \subseteq \alpha'}{\Gamma_1, \Gamma_2 \vdash e_1\ e_2 : \sigma_2}\textsc{App}$$

with $\alpha \subseteq \alpha$, $\times \subseteq \bullet$, and (sup) the corresponding supremum (least upper bound)

Subtyping

$$\frac{\Gamma \vdash e : \tau^{\bullet}}{\Gamma \vdash e : \tau^{\times}}\textsc{Unique}$$

for simplicity we treat the function space as invariant

Structural Rules

$$\frac{\Gamma, x : \tau^{\times}, x : \tau^{\times} \vdash e : \sigma}{\Gamma, x : \tau^{\times} \vdash e : \sigma}\textsc{Contr} \qquad \frac{\Gamma \vdash e : \sigma}{\Gamma, x : \sigma' \vdash e : \sigma}\textsc{Weak}$$

Fig. 1: Uniqueness Typing

Logical Rules

$$\frac{}{x : \sigma \vdash x : \sigma}\textsc{Var}$$

$$\frac{\Gamma, x : \sigma_1 \vdash e : \sigma_2 \quad \alpha \subseteq \sup \Gamma}{\Gamma \vdash \lambda x \cdot e : \sigma_1 \xrightarrow[\sup \Gamma]{\alpha} \sigma_2}\textsc{Abs} \qquad \frac{\Gamma_1 \vdash e_1 : \sigma_1 \xrightarrow[\alpha']{\alpha} \sigma_2 \quad \Gamma_2 \vdash e_2 : \sigma_1}{\Gamma_1, \Gamma_2 \vdash e_1\ e_2 : \sigma_2}\textsc{App}$$

Subtyping

$$\frac{\Gamma \vdash e : \tau^{\times}}{\Gamma \vdash e : \tau^{\bullet}}\textsc{Affine}$$

Structural rules as above.

Fig. 2: Affine Typing

Logical Rules

$$\frac{}{x : \sigma \vdash x : \sigma}\textsc{Var}$$

$$\frac{\Gamma, x : \sigma_1 \vdash e : \sigma_2 \quad \alpha \subseteq \sup \Gamma}{\Gamma \vdash \lambda x \cdot e : \sigma_1 \xrightarrow[\sup \Gamma]{\alpha} \sigma_2}\textsc{Abs} \qquad \frac{\Gamma_1 \vdash e_1 : \sigma_1 \xrightarrow[\alpha']{\alpha} \sigma_2 \quad \Gamma_2 \vdash e_2 : \sigma_1 \quad \alpha \subseteq \alpha'}{\Gamma_1, \Gamma_2 \vdash e_1\ e_2 : \sigma_2}\textsc{App}$$

Generalization and Instantiation

$$\frac{\Gamma \vdash e : \sigma\{{}^{\bullet}/_{a}\} \quad \Gamma \vdash e : \sigma\{{}^{\times}/_{a}\}}{\Gamma \vdash e : (\forall a \cdot \sigma)^{\sup \Gamma}}\textsc{Gen} \qquad \frac{\Gamma \vdash e : (\forall a \cdot \sigma)^{\alpha}}{\Gamma \vdash e : \sigma\{{}^{\alpha'}/_{a}\}}\textsc{Inst}$$

Structural rules as above.

Fig. 3: Unified using Polymorphism

two (identical) assumptions for x is a different typing environment to $\{x : \sigma\}$ containing a single assumption. Moreover, in rule App we *union* the typing environment used to type the function with the typing environment used to type the argument, rather than using the same environment for both.

This means that we must be explicit about *structural* operations on the typing environment. Rule Contr allows us to *contract* two typing assumptions, while rule Weak allows us to weaken a typing derivation by introducing an additional (unused) assumption. Importantly, Contr applies only to non-unique terms, so that we can derive

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{f : \tau^\times \to \tau^\times \to \sigma, x : \tau^\times \vdash f\ x : \tau^\times \to \sigma} \quad \dfrac{}{x : \tau^\times \vdash x : \tau^\times}}{f : \tau^\times \to \tau^\times \to \sigma, x : \tau^\times, x : \tau^\times \vdash f\ x\ x : \sigma}\ \textsc{App}}{f : \tau^\times \to \tau^\times \to \sigma, x : \tau^\times \vdash f\ x\ x : \sigma}\ \textsc{Contr}}{f : \tau^\times \to \tau^\times \to \sigma \vdash \lambda x \cdot f\ x\ x : \tau^\times \to \sigma}\ \textsc{Abs}}{\varnothing \vdash \lambda f \cdot \lambda x \cdot f\ x\ x : (\tau^\times \to \tau^\times \to \sigma) \to \tau^\times \to \sigma}\ \textsc{Abs}$$

but, crucially, we cannot find any derivation for

$$\varnothing \vdash \lambda f \cdot \lambda x \cdot f\ x\ x : (\tau^\bullet \to \tau^\bullet \to \sigma) \to \tau^\bullet \to \sigma$$

The restriction on the structural rule Contr is what makes uniqueness typing a *substructural* logic.

2.2 Subtyping

Uniqueness is a *guarantee* that a term is not shared; however, it is safe to ignore that guarantee. For instance, we can find a term with type

$$(\tau^\times \to \tau^\times \to \sigma) \to \tau^\bullet \to \sigma$$

(exercise: what is it?). For an example use case, consider a type `Array` of integer arrays with corresponding in-place updates

$$\begin{aligned}&\texttt{update} :: \texttt{Int} \to \texttt{Int} \to \texttt{Array}^\bullet \to \texttt{Array}^\bullet\\ &\texttt{sort} :: \texttt{Ascending} \to \texttt{Array}^\bullet \to \texttt{Array}^\bullet\\ &\ldots\end{aligned}$$

Once we are done with updating the array we can apply subtyping to get an $\texttt{Array}^\times$ which we can freely share but no longer update.

The combination of the restriction on Contr with subtyping (Unique) justifies reading "$\bullet$" as "non-shared".

2.3 Closure Typing

Consider a term such as

$$\lambda arr \cdot \lambda asc \cdot \texttt{sort}\ asc\ arr :: \texttt{Array}^\bullet \to \texttt{Ascending} \xrightarrow[\bullet]{\alpha} \texttt{Array}^\bullet$$

When we partially apply this function to a unique array, we get a function of type

$$\texttt{Ascending} \xrightarrow[\bullet]{\alpha} \texttt{Array}^\bullet$$

the annotation underneath the function arrow here means that this function has a unique term in its closure. It is important that this term is still unique when we (fully) apply the function, which is why rule App requires that when we apply a function with unique terms in its closure it must itself be unique. This is called uniqueness *propagation*, and is important whenever terms contain other terms (function closures, tuples, algebraic data types, etc.).

3 Affine Typing

Affine typing is a close cousin of uniqueness typing; the typing rules are shown in Fig. 2. Where "unique" can be interpreted as a *guarantee* that a term *has not* not shared, "affine" can be interpreted as a *restriction* that a term *may not* be shared, or, equivalently but more conventionally, can only be used once.

> *Aside.* Affine typing is closely related to linear typing, in which the weakening rule (Weak) is also limited to non-affine types. It is often claimed that such a type system guarantees that a term of linear type will be used "exactly" once; however, since linear type systems rarely guarantee the absence of divergence, this is a dubious claim. We will use "affine" throughout this paper as the more general term.

This duality between uniqueness typing and affine typing is evident in the typing rules too, in two ways. First, the subtyping relation is inverse (rule Affine). Where a guarantee of uniqueness can be forgotten but not invented, an affine restriction may be self-imposed but not ignored.

Second, like in uniqueness typing, when a closure contains a restricted term then that closure itself must be restricted; but unlike in uniqueness typing, that restriction must be enforced at the *definition* site (Abs) rather than the usage site (App). (Exercise: why is it unsafe to combine Unique subtyping with definition-site propagation, or Affine subtyping with usage-site propagation?)

For an example use case, consider a concurrent, impure (not referentially transparent) functional language with a type `Channel` of communication channels with corresponding functions

$$\texttt{send} :: \texttt{Int} \to \texttt{Channel}^\bullet \to \texttt{Unit}$$
$$\texttt{newChannel} :: \texttt{Unit} \to \texttt{Channel}^\times$$

We can pass a channel of type $\texttt{Channel}^\bullet$ to a thread, meaning that it can only send a single signal on the channel; a "master" thread can create a new channel using `newChannel`, spawn a number of slave threads, use subtyping to pass in an affine reference to this channel, and is then guaranteed that each slave thread will write at most once to the channel.

4 Polymorphism

Consider again the type of array update:

$$\texttt{update} :: \texttt{Int} \to \texttt{Int} \to \texttt{Array}^\bullet \to \texttt{Array}^\bullet$$

The *input* array must certainly be unique, but `update` does not itself care that the result is unique; that is, we could also provide

$$\texttt{update}' :: \texttt{Int} \to \texttt{Int} \to \texttt{Array}^\bullet \to \texttt{Array}^\times$$

Similarly, in the channel example, `send` took a restricted channel, but we could also provide

$$\texttt{send}' :: \texttt{Int} \to \texttt{Channel}^\times \to \texttt{Unit}$$

Using uniqueness subtyping we can define $\texttt{update}'$ in terms of `update`; using affine subtyping we can define $\texttt{send}'$ in terms of `send`. However, since the subtyping relation in both cases is so shallow, there is a more obvious generalization of both functions:

$$\begin{aligned}&\texttt{update} :: \forall a \cdot \texttt{Int} \to \texttt{Int} \to \texttt{Array}^\bullet \to \texttt{Array}^a\\ &\texttt{send} :: \forall a \cdot \texttt{Int} \to \texttt{Channel}^a \to \texttt{Unit}\end{aligned}$$

We only need a *single* construct to capture *both* subtyping relations, and thus we arrive at the central thesis of this paper: we can use polymorphism to combine uniqueness typing and affine typing within a single system. The only difference whether we use polymorphism in the codomain (`update`) or the domain of the function (`send`), once more establishing the duality between uniqueness and affine typing.

4.1 The polymorphic type system

We extend the systems of types with

$$\begin{aligned}\tau &::= c \mid \sigma \underset{\alpha}{\longrightarrow} \sigma' \mid \forall a \cdot \sigma\\ \sigma &::= \tau^\alpha \qquad\qquad \text{(as before)}\end{aligned}$$

A function of type $\tau_1^\bullet \xrightarrow{\bullet} \tau_2^\times$ is a *unique* function from a unique τ_1 to a non-unique τ_2; the attribute on the function is distinct from the attributes on its domain and codomain. Similarly, a value of type, say, $(\forall a.\texttt{Array}^a)^\bullet$ is a *unique*

polymorphic value that can be instantiated to a unique or non-unique array. The uniqueness on the polymorphic value itself means that it can only be instantiated once. The analogy with functions is appropriate: a polymorphic value can be interpreted as a function that takes a type argument; and like functions, polymorphic values must be unique themselves if they have any unique elements in their "closure".

The rules for the polymorphic type system are shown in Fig. 3. The structural rules are as they were in uniqueness typing and affine typing. We no longer have subtyping, however; this means that the "$\bullet$" annotation means "has *and* may never be shared", thus no longer distinguishing between "unique" and "affine"; we can choose either interpretation based on the application we have in mind. Rule Gen embodies the propagation we described in the previous section. Rule Inst is the familiar instantiation rule, where we ignore the attribute on the polymorphic value itself.

Propagation for functions is now enforced in *both* definition *and* usage sites (Abs and App). This is overkill; it would suffice to enforce propagation in Abs (and do away with closure typing completely): after all, in the absence of subtyping if a function is unique when it is created it must still be unique when it applied. Formally proving that this simpler type system is equivalent to the one we have presented, however, is slightly non-trivial and non-essential to the central message of this paper. We chose the representation in Fig. 3 to aid the comparison to the uniqueness and affine typing systems; we do not necessarily suggest to use the type system in this particular form.

When introducing a new type system, two questions arise:

- Is the new type system *sound*? That is, are there any programs accepted by the type system that should not be?
- Is the new type system *complete*? That is, are there any programs not accepted by the type system that should be?

We will show in Sect. 4.2 that the polymorphic system is *relatively sound*: it does not accept any more programs than the intersection of uniqueness typing and affine typing does. Few type systems can claim to be complete, and ours is no exception. In fact, even relative completeness fails, but we will argue in Sect. 4.3 that the loss is outweighed by the benefits.

4.2 Soundness

We show that if a program e is accepted by the polymorphic type system (i.e., there exists Γ, σ such that $\Gamma \vdash e : \sigma$) then it is also accepted by both the unique and affine systems. We show this by providing a translation $\lfloor \sigma \rfloor$ from the polymorphic type system to the unique or affine type system. We consider the case for the unique type system first. We translate polymorphism to uniqueness:

Definition 1 (Translation from polymorphic to unique types).

$$\begin{aligned}\lfloor c^\alpha \rfloor &= c^\alpha \\ \lfloor \sigma_1 \xrightarrow[\alpha']{\alpha} \sigma_2 \rfloor &= \lfloor \sigma_1 \rfloor \xrightarrow[\alpha']{\alpha} \lfloor \sigma_2 \rfloor \\ \lfloor (\forall a \cdot \sigma)^\alpha \rfloor &= \lfloor \sigma \{ {}^{\bullet}/_{a} \} \rfloor\end{aligned}$$

This translation extends in the obvious manner to typing environments.

Proposition 1 (Soundness wrt to uniqueness typing).
If $\Gamma \vdash e : \sigma$ *then* $\lfloor \Gamma \rfloor \vdash e : \lfloor \sigma \rfloor$.

Proof. By induction on $\Gamma \vdash e : \sigma$. The logical and structural rules are straightforward. For Gen the conclusion follows from the induction hypothesis at the first premise. For Inst it follows from Unique (or immediately). □

As expected, the case for the affine type system is similar, but dual: we translate polymorphism to unrestricted (non-affine).

Definition 2 (Translation from polymorphic to affine types).

$$\begin{aligned}\lfloor c^\alpha \rfloor &= c^\alpha \\ \lfloor \sigma_1 \xrightarrow[\alpha']{\alpha} \sigma_2 \rfloor &= \lfloor \sigma_1 \rfloor \xrightarrow[\alpha']{\alpha} \lfloor \sigma_2 \rfloor \\ \lfloor (\forall a \cdot \sigma)^\alpha \rfloor &= \lfloor \sigma \{ {}^{\times}/_{a} \} \rfloor\end{aligned}$$

Proposition 2 (Soundness wrt to affine typing).
If $\Gamma \vdash e : \sigma$ *then* $\lfloor \Gamma \rfloor \vdash e : \lfloor \sigma \rfloor$.

Proof. Like the proof of Prop. 1, but using the *second* premise of Gen and using Affine instead of Unique. □

4.3 Completeness

In uniqueness typing we can create non-unique functions with unique elements in their closure, even though we can no longer *apply* those functions. Likewise, in affine typing we can apply non-unique functions with unique elements in their closure, even though we can never create such functions. Neither is possible in the polymorphic system, which means that the polymorphic system is not relatively complete with respect to either the uniqueness or affine type systems.

We can give a partial completeness result, however. Define the following lifting from the monomorphic types into the polymorphic type system:

Definition 3 (Lifting types).

$$\begin{aligned}\lceil c^\alpha \rceil &= c^\alpha \\ \lceil \sigma_1 \xrightarrow[\alpha'']{\alpha'} \sigma_2 \rceil &= \lceil \sigma_1 \rceil \xrightarrow[\alpha]{\alpha} \lceil \sigma_2 \rceil \ \textit{where}\ \alpha = \sup\{a', a''\}\end{aligned}$$

Proposition 3 (Partial completeness). *If* $\Gamma \vdash^{unique} e : \sigma$ *or* $\Gamma \vdash^{affine} e : \sigma$ and the typing derivation does not rely on subtyping *then* $\Gamma \vdash^{poly} e : \sigma$.

Proof. Two separate straightforward induction proofs. □

In other words, programs that do not rely on subtyping will be accepted by the polymorphic type system, too. Most applications of subtyping can be replaced by use of polymorphism, as we saw at the start of Section 4. That is, for uniqueness typing we can translate

$$[\![\sigma \to \tau^\bullet]\!]_{\text{unique}} = \forall a \cdot [\![\sigma]\!] \to [\![\tau^a]\!]$$

Similarly, for affine typing we can translate

$$[\![\tau^\times \to \sigma]\!]_{\text{affine}} = \forall a \cdot [\![\tau^a]\!] \to [\![\sigma]\!]$$

(note again the duality: $\bullet$ *vs* $\times$, codomain *vs* domain). In both cases subtyping can then be replaced by instantiation.

This translation is not entirely uniform, however. As we mentioned at the start of this section, the following types are not inhabited, even though their corresponding unique or affine types are:

$$\begin{aligned}[\![\tau_1^\bullet \to \tau_2^\times \xrightarrow[\bullet]{\bullet} \tau_1^\bullet]\!]_{\text{unique}} &= \forall a \cdot \tau_1^\bullet \to \tau_2^\times \xrightarrow[\bullet]{a} \tau_1^\bullet \\ [\![(\sigma_1 \xrightarrow[\bullet]{\times} \sigma_2) \to \sigma_1 \to \sigma_2]\!]_{\text{affine}} &= \forall a \cdot (\sigma_1 \xrightarrow[\bullet]{a} \sigma_2) \to \sigma_1 \to \sigma_2\end{aligned}$$

Note that the use of subtyping is not essential for dealing with "observing" terms at non-unique types; instead, we need a special typing rule for strict-let, and preferably some way of making sure that these non-unique terms don't escape the strict-let. See [10, Section 2.8.9] for more details.

4.4 Example Application

In Sect. 3 we used affine types to restrict how often a thread could write to a channel. We mentioned that this was in the context of an "impure" language, because affine types cannot be used to model side effects[1] (we need uniqueness typing, instead). However, now that we have both, we could define[2]

$$\begin{aligned}&\texttt{withNewChannel} :: \texttt{World}^\bullet \to (\texttt{World}^\bullet \to (\forall a \cdot \texttt{Channel}^a)^\times \to \sigma) \xrightarrow[\bullet]{\bullet} \sigma \\ &\texttt{send} :: \forall a \cdot \texttt{Int} \to \texttt{World}^\bullet \to \texttt{Channel}^a \xrightarrow[\bullet]{\bullet} \texttt{World}^\bullet\end{aligned}$$

If we applied the translation from Sect. 4.3 indiscriminately we would have used a unique world in negative position ("input") and a polymorphic world in positive

[1] Wadler, one of the big proponents of linear logic, states [14]: "Does this mean that linearity is useless for practical purposes? Not completely. Dereliction [subtyping] 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".

[2] We could return a pair instead of using continuation passing style, but we have not covered products in this paper.

position ("output"). However, realistically we never want to share the world so we can simplify the types and not use polymorphism.

The situation for channels is a little different. We have used a polymorphic value in negative position, since we can send on channels we are allowed use once just as well as we can send on unrestricted channels. In `withChannel` we use a value of type $(\forall a \cdot \texttt{Channel}^a)^\times$ in positive position; the continuation (the "master thread") can use this polymorphic value as often as it wants to create "affine" (use-once) channels for the slave threads and then instantiate it at an unrestricted value itself to read the values written by all the slave threads.

We thus switch back and forth between the two interpretations at will, and use instantiation in the place of subtyping: we use polymorphism to model unique and affine typing.

5 Related Work

Uniqueness typing was introduced in [2] and implemented in the pure functional programming language Clean [7]; variations have been implemented in SAC [8] and Mercury [5, 6].

The version we presented in Fig. 1 differs significantly from the Clean type system, however. Clean does not use closure typing, which was introduced in the context of uniqueness typing in [12]. Instead, Clean regards some unique types (in particular, function types) as *necessarily* unique: subtyping does not apply to them. This makes it possible to enforce propagation at definition site, rather like in our polymorphic system. However, this non-uniformity of the subtyping relation results in a loss of principal types. For instance, we have

$$\lambda x \cdot (x, x) :: \texttt{Array}^\bullet \to (\texttt{Array}^\times, \texttt{Array}^\times)$$

and

$$\lambda x \cdot (x, x) :: (\sigma_1 \xrightarrow{\times} \sigma_2) \to (\sigma_1 \xrightarrow{\times} \sigma_2, \sigma_1 \xrightarrow{\times} \sigma_2)$$

but no more general type that can be instantiated to both. Note that the polymorphic system in Fig. 3 does not satisfy principal types either. For instance, given a function $f :: \forall a, a' \cdot \tau^a \to \tau^{a'} \xrightarrow{a} \sigma$, we have

$$\lambda x \cdot \lambda y \cdot \lambda z \cdot f\ x\ y :: \tau^\times \xrightarrow{\times} \tau^\times \xrightarrow{\times} \sigma' \xrightarrow{\times} \sigma$$

$$\lambda x \cdot \lambda y \cdot \lambda z \cdot f\ x\ y :: \tau^\times \xrightarrow{\times} \tau^\bullet \xrightarrow{\times} \sigma' \xrightarrow{\bullet} \sigma$$

$$\lambda x \cdot \lambda y \cdot \lambda z \cdot f\ x\ y :: \tau^\bullet \xrightarrow{\times} \tau^\times \xrightarrow{\bullet} \sigma' \xrightarrow{\bullet} \sigma$$

and

$$\lambda x \cdot \lambda y \cdot \lambda z \cdot f\ x\ y :: \tau^\bullet \xrightarrow{\times} \tau^\bullet \xrightarrow{\bullet} \sigma' \xrightarrow{\bullet} \sigma$$

but no more general type that captures all four; in particular, although we have

$$\lambda x \cdot \lambda y \cdot \lambda z \cdot f\ x\ y :: \forall a, a' \cdot \tau^a \xrightarrow{\times} \tau^{a'} \xrightarrow{a} \sigma' \xrightarrow{\bullet} \sigma$$

the annotation on the final arrow must be "$\bullet$" because we cannot express within the type system that it must be unique if either a or a' is and thus we must

be conservative. One way to solve this problem is introduce boolean expressions as annotations [13]; this is a nice approach because boolean unification is well-understood and hence can we use standard type inference algorithms for such a type system (it might also be possible to lift the "sup" operation we used to the type level).

Linear logic was introduced by Girard [3]; its use as a type system was pioneered by Wadler [14, 9]. Several authors have proposed type systems that explicitly combine uniqueness typing and linear typing [4, 1, 11]. All of these systems however have *explicit* notions of uniqueness and affinity, rather than using one concept to model both.

The author's PhD thesis contains a detailed review of these and other papers [10].

6 Conclusions

Uniqueness typing and affine or linear typing are dual type systems. Uniqueness gives a *guarantee* that an term *has not* been shared, thus enabling destructive update and modelling of side effects in a pure functional language. Affinity imposes a *restriction* that a term *may not* be shared, thus enabling more precise APIs (a continuation that can be invoked at most once, a channel that can be sent on at most once, etc.). Both type systems have different purposes and indeed it is useful to combine them.

In this paper we have shown that when we introduce polymorphism—a useful construct in its own right—we do not need to distinguish explicitly between uniqueness and affinity anymore, but enable the programmer to choose between either interpretation by introducing polymorphism in negative or positive positions. For instance, we saw that we might type destructive array updates as

$$\texttt{update} :: \forall a \cdot \texttt{Int} \to \texttt{Int} \to \texttt{Array}^{\bullet} \to \texttt{Array}^{a}$$

Other API choices are possible too, of course. For instance, if non-updatable arrays have a more efficient representation in memory then we might want to change the API to

$$\begin{aligned}&\texttt{update} :: \texttt{Int} \to \texttt{Int} \to \texttt{Array}^{\bullet} \to \texttt{Array}^{\bullet}\\&\texttt{freeze} :: \texttt{Array}^{\bullet} \to \texttt{Array}^{\times}\end{aligned}$$

By replacing subtyping with polymorphism we place this choice in the hands of the API designer.

The particular type system that we presented was designed to aid the comparison with "traditional" uniqueness and affine type systems. It can be simplified and extended in various ways. At the very least one might want universal quantification over base types (τ) as well as type attributes (α); it is possible to use a kind system to use a single construct for both [13]. As mentioned in Sect. 5, we can introduce boolean expressions as type attributes in order to obtain principal types. Chapter 8 of [10] contains many more avenues for future work.

Acknowledgements This paper is a follow-up from the author's PhD thesis on uniqueness typing, which Rinus Plasmeijer mentored. Rinus, your support and enthusiasm was greatly appreciated.

References

1. Ahmed, A., Fluet, M., Morrisett, G.: A step-indexed model of substructural state. In: Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming (ICFP). pp. 78–91. ACM (2005)
2. Barendsen, E., Smetsers, S.: Uniqueness typing for functional languages with graph rewriting semantics. Mathematical Structures in Computer Science 6, 579–612 (1996)
3. Girard, J.Y.: Linear logic. Theoretical Computer Science 50(1), 1–102 (1987)
4. Hage, J., Holdermans, S., Middelkoop, A.: A generic usage analysis with subeffect qualifiers. In: Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP). pp. 235–246. ACM (2007)
5. Henderson, F.: Strong modes can change the world! (Nov 1992), honours Report, Department of Computer Science, University of Melbourne
6. Overton, D.: Precise and expressive mode systems for typed logic programming languages. Ph.D. thesis, The University of Melbourne (Dec 2003)
7. Plasmeijer, R., van Eekelen, M.: Clean Language Report (version 2.1) (Nov 2002)
8. Scholz, S.B.: Single assignment C: efficient support for high-level array operations in a functional setting. Journal of Functional Programming 13(6), 1005–1059 (2003)
9. Turner, D.N., Wadler, P., Mossin, C.: Once upon a type. In: Proceedings of the 7th international conference on Functional Programming languages and Computer Architecture (FPCA). pp. 1–11. ACM (1995)
10. de Vries, E.: Making Uniqueness Typing Less Unique. Ph.D. thesis, Trinity College Dublin (2008)
11. de Vries, E., Francalanza, A., Hennessy, M.: Uniqueness typing for resource management in message-passing concurrency. Journal of Logic and Computation (2012)
12. de Vries, E., Plasmeijer, R., Abrahamson, D.M.: Uniqueness typing redefined. In: Horváth, Z., Zsók, V., Butterfield, A. (eds.) Proceedings of the 18th international symposium on Implementation and Application of Functional Languages (IFL). Lecture Notes in Computer Science, vol. 4449, pp. 181–198. Springer-Verlag (2007)
13. de Vries, E., Plasmeijer, R., Abrahamson, D.M.: Uniqueness typing simplified. In: Chitil, O., Horváth, Z., Zsók, V. (eds.) Proceedings of the 19th international symposium on Implementation and Application of Functional Languages (IFL). Lecture Notes in Computer Science, vol. 5083, pp. 181–198. Springer-Verlag (2008)
14. Wadler, P.: Is there a use for linear logic? In: Proceedings of the 2nd ACM SIGPLAN symposium on Partial Evaluation and semantics-based program manipulation (PEPM). pp. 255–273. ACM (1991)