Practical Affine Types

Jesse A. Tov Riccardo Pucella
Northeastern University, Boston, Massachusetts, USA
{tov,riccardo}@ccs.neu.edu

Abstract

Alms is a general-purpose programming language that supports practical affine types. To offer the expressiveness of Girard's linear logic while keeping the type system light and convenient, Alms uses expressive kinds that minimize notation while maximizing polymorphism between affine and unlimited types.

A key feature of Alms is the ability to introduce abstract affine types via ML-style signature ascription. In Alms, an interface can impose stiffer resource usage restrictions than the principal usage restrictions of its implementation. This form of sealing allows the type system to naturally and directly express a variety of resource management protocols from special-purpose type systems.

We present two pieces of evidence to demonstrate the validity of our design goals. First, we introduce a prototype implementation of Alms and discuss our experience programming in the language. Second, we establish the soundness of the core language. We also use the core model to prove a principal kinding theorem.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Languages

Keywords Affine types, linear logic, type systems, polymorphism, modules

1. A Practical Affine Type System

Alms is a practical, general-purpose programming language with affine types. *Affine types* enforce the discipline that some values are not used more than once, which in Alms makes it easy to define new, resource-aware abstractions. *General-purpose* means that Alms offers a full complement of modern language features suitable for writing a wide range of programs. *Practical* means that Alms is neither vaporware nor a minimal calculus—it is possible to download Alms today and try it out.

Rationale. Resource-aware type systems divide into two camps: foundational calculi hewing closely to linear logic, and implementations of special-purpose type systems designed to solve special problems. We argue that a general, practical type system based on Girard's linear logic (1987) can naturally and directly express many of the special cases, such as region-based memory management, aliasing control, session types, and typestate. To this end, the language must satisfy several desiderata:

PoPL'11, January 26–28, 2011, Austin, Texas, USA.


```
module type RW_LOCK = sig
  type (α, β) array    — array of α identified by β
  type excl            — exclusive access
  type shared          — shared access
  type β@γ : A         — grants γ-level access to array β

  val new      : int → α → ∃β. (α, β) array

  val acquireW: (α, β) array → β@excl
  val acquireR : (α, β) array → β@shared
  val release  : (α, β) array → β@γ → unit

  val set: (α, β) array → int → α → β@excl → β@excl
  val get: (α, β) array → int → β@γ → α × β@γ
end
```

Figure 1. An interface for reader-writer locks (§2)

Convenience. Unlimited values are the common case, so working with them is as smooth as in a conventional language.

Expressiveness. A wide variety of resource-aware type systems appear naturally as idioms.

Familiarity. It is easy to use and understand.

Pragmatics. It provides the trappings of a modern, high-level programming language, such as algebraic datatypes, pattern matching, exceptions, concurrency, and modules.

Soundness. It has a clear theoretical foundation.

We show that Alms meets these criteria.

Alms employs a dependent kind system to determine whether a particular type is affine or unlimited and to support polymorphism over affine and unlimited types. This approach may sound complicated, but in practice it is no stranger or harder to understand than the type systems of other functional programming languages.

Affine types, a weakening of linear types, forbid duplication of some values; unlike with linear types, all values may be dropped. This flexibility is appropriate to a high-level, garbage-collected language, and it interacts better with other features such as exceptions.

Our Contributions. This paper introduces the programming language Alms, its implementation, and its basic theory:

- We describe the design of Alms, whose novel features include precise kinds for affine type constructors, and demonstrate how it expresses a variety of resource-aware idioms (§2).
- Our implementation is a usable, full-featured prototype in which we have written several thousand lines of code (§3).
- Alms rests on a firm theoretical basis. We provide a formal model (§4) and establish essential theoretical properties (§5).

Our implementation and full proofs are available at www.ccs.neu.edu/~tov/pubs/alms.

2. Learn Alms in 2.5 Pages

Alms is a typed, call-by-value, impure functional language with algebraic data types, pattern matching, reference cells, threads, exceptions, and modules with opaque signature ascription. Alms reads like Ocaml (Leroy et al. 2008) but is explicitly typed. In most cases local type inference renders explicit type instantiation unnecessary.

We introduce Alms through a series of examples. Consider a simple Alms function, *deposit*, that updates one element of an array by adding an integer:

```
let deposit (a: int Array.array) (acct: int) (amount: int) =
  Array.set a acct (Array.get a acct + amount)
```

This function has a race condition between the read and the write, so we may want to use a lock to enforce mutual exclusion:

```
let deposit (a: int Array.array) (acct: int)
            (amount: int) (lock: Lock.lock) =
  Lock.acquire lock;
  Array.set a acct (Array.get a acct + amount);
  Lock.release lock
```

Affine data. Locks can ensure mutual exclusion, but using them correctly is error-prone. A rather coarse alternative to ensure mutual exclusion is to forbid aliasing of the array altogether. If we have the only reference to an array then no other process can operate on it concurrently. In Alms, we do this by declaring an interface that includes a new, abstract array type:

```
module type AF_ARRAY = sig
  type α array : A

  val new : int → α → α array
  val set  : α array → int -A→ α -A→ α array
  val get  : α array → int -A→ α × α array
end

module AfArray : AF_ARRAY = struct ... end
```

The notation ": **A**" specifies that type α AfArray.array has kind **A**, as in *affine*, which means that any attempt to duplicate a reference to such an array is a type error. Two points about the types of *AfArray.get* and *AfArray.set* are worth noting:

- Each must return an array because the caller cannot reuse the reference to the array supplied as an argument.
- Type $\tau_1 \xrightarrow{\mathsf{A}} \tau_2$ has kind **A**, which means that it may be used at most once.[1] This is necessary because reusing a function partially applied to an affine value would reuse that value.

We now rewrite *deposit* to use the AF_ARRAY interface:

```
let deposit (a: int AfArray.array) (acct: int) (amt: int) =
  let (balance, a) = AfArray.get a acct in
    AfArray.set a acct (balance + amt)
```

If we attempt to use an AfArray.array more than once without single-threading it, a type error results:

```
let deposit (a: int AfArray.array) (acct: int) (amt: int) =
  let (balance,_) = AfArray.get a acct in
    AfArray.set a acct (balance + amt)
```

Alms reports that the affine variable *a* is duplicated.

Implementing *AfArray* is just a matter of wrapping the primitive array type and operations, and sealing the module with an opaque signature ascription:

```
module AfArray : AF_ARRAY = struct
  type α array = α Array.array

  let new = Array.new
  let set (a: α array) (ix: int) (v: α) = Array.set a ix v; a
  let get (a: α array) (ix: int) = (Array.get a ix, a)
end
```

The original array type α Array.array has kind **U**, as in *unlimited*, because it places no limits on duplication. We can use it to represent an abstract type of kind **A**, however, because **U** is a subkind of **A**, and Alms's kind subsumption rule allows assigning an abstract type a greater kind than that of its concrete representation. This is somewhat akin to Standard ML's treatment of equality types (Milner et al. 1997) and Ocaml's treatment of type constructor variance (Leroy et al. 2008). In SML, eqtype is subsumed by type, in that signature matching can abstract an equality type to a non-equality type but not vice versa.

We need not change *new* at all, and *get* and *set* are modified slightly to return the array as required by the interface.

Affine capabilities. The affine array interface is quite restrictive. Because it requires single-threading an array through the program, it cannot support operations that do not actually require exclusive access to the array. However, Alms supports creating a variety of abstractions to suit our needs. One way to increase our flexibility is to separate the reference to the array from the *capability* to read and write the array. Only the latter needs to be affine.

For example, we may prefer an interface that supports "dirty reads," which do not require exclusive access but are not guaranteed to observe a consistent state:

```
module type CAP_ARRAY = sig
  type (α, β) array
  type β cap : A

  val new: int → α → ∃β. (α, β) array × β cap
  val set  : (α, β) array → int → α → β cap → β cap
  val get  : (α, β) array → int → β cap → α × β cap

  val dirtyGet : (α, β) array → int → α
end
```

In this signature, (α, β) array is now unlimited and β cap is affine. Type array's second parameter, β, is a "stamp" used to tie it to its capability, which must have type β cap (where β matches). In particular, the type of *new* indicates that it returns an existential containing an array and a capability with matching stamps. The existential guarantees that the stamp on an array can only match the stamp on the capability created by the same call to *new*.

Operations *set* and *get* allow access to an array only when presented with the matching capability. This ensures that *set* and *get* have exclusive access with respect to other *set*s and *get*s. They no longer return the array, but they do need to return the capability. On the other hand, *dirtyGet* does not require a capability and should not return one.

For example, the CAP_ARRAY interface allows us to shuffle an array while simultaneously computing an approximate sum:

```
let shuffleAndDirtySum (a: (α, β) CapArray.array)
                       (cap: β CapArray.cap)      =
  let th1 = Thread.fork (λ_ → inPlaceShuffle a cap) in
  let th2 = Thread.fork (λ_ → dirtySumArray a) in
    (Thread.wait th1, Thread.wait th2)
```

To implement CAP_ARRAY, we need suitable representations for its two abstract types. We represent CAP_ARRAY's arrays by the primitive array type, and capabilities by type unit, which is adequate because these capabilities have no run-time significance.

[1] It is tempting to call this an affine function, but standard terminology says that an affine function uses its *argument* at most once, whereas here we have the type of a function *itself* usable at most once. Whether an Alms function is affine is determined by the kind of the type of its formal parameter.

```
module A = Array

module CapArray : CAP_ARRAY = struct
  type (α, β) array = α A.array
  type β cap        = unit

  let new (size: int) (init: α) = (A.new size init, ())
  let set (a: α A.array) (ix: int) (v: α) _ = A.set a ix v
  let get (a: α A.array) (ix: int) _ = (A.get a ix, ())
  let dirtyGet = A.get
end
```

Type unit has kind **U**, but as in the previous example, we can abstract it to **A** to match the kind of β CapArray.cap. The implementation of the operations is in terms of the underlying array operations, with some shuffling to ignore capability arguments (in *set* and *get*) and to construct tuples containing () to represent the capability in the result (in *new* and *get*).[2]

Capabilities are values. Capabilities such as β CapArray.cap often represent the state of a resource, but in Alms they are also ordinary values. They may be stored in immutable or mutable data structures, packed into exceptions and thrown, or sent over communication channels like any other value. For example, suppose we would like a list of array capabilities. Lists are defined thus in the standard library:

```
type α̂ list = Nil | Cons of α̂ × α̂ list
```

The type variables we have seen until now could only be instantiated with unlimited types, but the diacritic on type variable $\hat{\alpha}$ indicates that $\hat{\alpha}$ may be instantiated to any type, whether affine or unlimited.

Whether a list should be treated as affine or unlimited depends on whether the contents of the list is affine or unlimited. Alms represents this fact by giving the list type constructor a dependent kind, where kind $\langle\hat{\alpha}\rangle$ denotes the kind of $\hat{\alpha}$:

$\hat{\alpha}$ list : $\langle\hat{\alpha}\rangle$ — list *has kind* $\Pi\hat{\alpha}.\,\langle\hat{\alpha}\rangle$

That is, the kind of a list is the same as the kind of its element type: type int list has kind **U**, whereas β CapArray.cap list has kind **A**.

In general, the kind of a type is the least upper bound of the kinds of the types that occur directly in its representation. For example:

type $(\hat{\alpha},\hat{\beta})\,\mathsf{r} = \hat{\alpha}\times\hat{\beta}$	— $\langle\hat{\alpha}\rangle \sqcup \langle\hat{\beta}\rangle$
type $(\hat{\alpha},\hat{\beta})\,\mathsf{s} = \mathsf{int}\times\hat{\beta}$	— $\langle\hat{\beta}\rangle$
type $(\hat{\alpha},\hat{\beta})\,\mathsf{t} = T_1$ **of** $\hat{\alpha}$ | T_2 **of** $(\hat{\beta},\hat{\alpha})\,\mathsf{t}$	— $\langle\hat{\alpha}\rangle \sqcup \langle\hat{\beta}\rangle$
type $(\hat{\alpha},\hat{\beta})\,\mathsf{u} = U_1$ | U_2 **of** $(\hat{\beta},\hat{\alpha})\,\mathsf{u}$	— **U**
type $(\hat{\alpha},\hat{\beta})\,\mathsf{v} = \hat{\alpha}\times(\mathsf{unit}\to\hat{\beta})$	— $\langle\hat{\alpha}\rangle$
type $(\hat{\alpha},\hat{\beta})\,\mathsf{w} = \hat{\alpha}\times(\mathsf{unit}\xrightarrow{\langle\hat{\beta}\rangle}\mathsf{unit})$	— $\langle\hat{\alpha}\rangle \sqcup \langle\hat{\beta}\rangle$

Because both $\hat{\alpha}$ and $\hat{\beta}$ are part of the representation of $(\hat{\alpha},\hat{\beta})\,\mathsf{r}$, it must be affine if either of its parameters is affine. On the other hand, the phantom parameter $\hat{\alpha}$ is not part of the representation of $(\hat{\alpha},\hat{\beta})\,\mathsf{s}$, so that has kind $\langle\hat{\beta}\rangle$. The kinds of t and u are the least solutions to these inequalities:

$$\kappa((\hat{\alpha},\hat{\beta})\,\mathsf{t}) \sqsupseteq \kappa(\hat{\alpha}) \qquad \kappa((\hat{\alpha},\hat{\beta})\,\mathsf{u}) \sqsupseteq \mathbf{U}$$
$$\kappa((\hat{\alpha},\hat{\beta})\,\mathsf{t}) \sqsupseteq \kappa((\hat{\beta},\hat{\alpha})\,\mathsf{t}) \qquad \kappa((\hat{\alpha},\hat{\beta})\,\mathsf{u}) \sqsupseteq \kappa((\hat{\beta},\hat{\alpha})\,\mathsf{u})$$

The kind of each type must be at least as restrictive as the kinds of all of its alternatives.

For $(\hat{\alpha},\hat{\beta})\,\mathsf{v}$, the kind of $\hat{\beta}$ does not appear because the domain and codomain of a function type are not part of the function's representation. Instead, function types have their kind in a superscript as in the definition of type w. We saw this in the *AfArray* example, where the superscripts were all **A**. (When the kind is **U**, we often omit it.) We discuss the kinds of function types and their subtyping in more depth in §4.3.

More possibilities. The rules of Alms are flexible enough to express a wide variety of designs. For example, the ability to store capabilities in data structures allows us to create a more dynamic interface than the static capabilities of *CapArray*:

```
module type CAP_LOCK_ARRAY = sig
  include CAP_ARRAY
  val new     : int → α → ∃β.(α, β) array
  val acquire : (α, β) array → β cap
  val release : (α, β) array → β cap → unit
end
```

This signature changes the type of *new* to return an array (with unique tag β) but no capability. To operate on the array, one needs to request a capability using *acquire*. Subsequent attempts to acquire a capability for the same array block until the capability is *release*d.

We implement CAP_LOCK_ARRAY in terms of *CapArray* without any privileged knowledge about the representation of its capabilities. The implementation relies on mvars, synchronized variables based Id's *M-structures* (Barth et al. 1991). An $\hat{\alpha}$ mvar may hold a value of type $\hat{\alpha}$ or it may be *empty*. While an mvar may contain an affine value, the mvar itself is always unlimited. This is safe because calling *take* on a non-empty mvar removes the value and returns it, while *take* on an empty mvar blocks until another thread *put*s a value in it.

To implement CAP_LOCK_ARRAY we now represent an array as a pair of the underlying (α,β) CapArray.array and an mvar to store its capability:

```
module CapLockArray : CAP_LOCK_ARRAY = struct
  module A = CapArray
  type (α, β) array = (α, β) A.array × β cap MVar.mvar

  let new (size: int) (init: α) =
    let (β, a, cap) = A.new size init in
      (a, MVar.new cap)

  let acquire ((_, mvar): (α, β) array) = MVar.take mvar
  let release ((_, mvar): (α, β) array) (cap: β cap) =
    MVar.put mvar cap

  let set ((a,_): (α, β) array) = A.set a
  let get ((a,_): (α, β) array) = A.get a
  ...
end
```

The *new* operation creates a new array-capability pair and stores the capability in an mvar. Operations *acquire* and *release* use the mvar component of the representation, while the old operations such as *set* must be lifted to project out the underlying CapArray.array.

There are many more possibilities. Figures 1 and 2 show two interfaces for reader-writer locks, which at any one time allow either exclusive read-write access or shared read-only access. Signature RW_LOCK (figure 1 on the first page) describes dynamic reader-writer locks. The signature declares nullary types excl and shared and an affine, binary type constructor $(\cdot\ @\ \cdot)$. Capabilities now have type $\beta@\gamma$, where β ties the capability to a particular array and γ records whether the lock is exclusive or shared. Operation *set* requires an exclusive lock ($\beta@$excl), but *get* allows γ to be shared or excl.

Signature FRACTIONAL (figure 2) describes static reader-writer locks based on fractional capabilities (Boyland 2003). As in the previous example, the capability type (β,γ) cap has a second parameter, which in this case represents a fraction of the whole

[2] If you think this unit shuffling is unnecessary, we agree (§7).

```
module type FRACTIONAL = sig
  type (α, β) array
  type 1
  type 2
  type γ/δ
  type (β, γ) cap : A

  val new : int → α → ∃β. (α, β) array × (β, 1) cap

  val split : (β, γ) cap → (β, γ/2) cap × (β, γ/2) cap
  val join  : (β, γ/2) cap × (β, γ/2) cap → (β, γ) cap

  val set  : (α, β) array → int → α →
               (β, 1) cap → (β, 1) cap
  val get  : (α, β) array → int →
               (β, γ) cap → α × (β, γ) cap
end
```

Figure 2. Another interface for reader-writer locks

capability. The fraction is represented using type constructors 1, 2, and $(\cdot\ /\ \cdot)$. A capability of type $(\beta, 1)$ cap grants exclusive access to the array with tag β, while a fraction less than 1 such as $1/2$ or $1/2/2$ indicates shared access. There are operations *split*, which splits a capability whose fraction is γ into two capabilities of fraction $\gamma/2$, and *join*, which combines two $\gamma/2$ capabilities back into one γ capability. Again, *set* requires exclusive access but *get* does not.

Syntax matters. Given *CapLockArray* as defined above, we can rewrite *deposit* to take advantage of it:

```
open CapLockArray

let deposit (a: (int, β) array) (acct: int) (amt: int) =
  let cap             = acquire a in
  let (balance, cap)  = get a acct cap in
  let cap             = set a acct (balance + amt) cap in
    release a cap
```

While this gets the job done, the explicit threading of the capability can be inconvenient and hard to read. To address this, Alms provides preliminary support for an alternative syntax inspired by a proposal by Mazurak et al. (2010):

```
let deposit (a: (int, β) array) (acct: int) (amt: int) =
  let !cap = acquire a in
    set a acct (get a acct cap + amt) cap;
    release a cap
```

The pattern **!***cap* bound by **let** marks *cap* as an "imperative variable," which means that within its scope, functions applied to *cap* are expected to return a pair of their real result and the new version of *cap*. Alms transforms this code into the explicitly-threaded version above. Currently this transformation happens before type checking, which means that it cannot compromise soundness but also cannot exploit type information.

3. Implementation and Experience

Our prototype, implemented in 16k lines of Haskell, is available at www.ccs.neu.edu/~tov/pubs/alms. Besides a usable interpreter, it includes all the example code from this paper and other Alms examples illustrating a variety of concepts:

- A capability-based interface to Berkeley Sockets ensures that the protocol to set up a socket is followed correctly. This library also shows how exceptions may be used for error recovery in the presence of affine capabilities.
- An echo server is built on top of the socket library.
- Two session types (Gay et al. 2003) libraries demonstrate different approaches to alternation: one uses anonymous sums and the other uses algebraic datatypes for named branches.
- Our version of Sutherland and Hodgman (1974) re-entrant polygon clipping uses session types to connect stream transducers.
- The Alms standard library implements higher-order coercions for checked downcasts which can, for example, turn a function of type $(\mathsf{unit} \xrightarrow{\mathsf{U}} \mathsf{unit}) \xrightarrow{\mathsf{U}} \mathsf{thread}$ into a function of type $(\mathsf{unit} \xrightarrow{\mathsf{A}} \mathsf{unit}) \xrightarrow{\mathsf{U}} \mathsf{thread}$ by adding a dynamic check.

These examples are not the last word on what can be done in a language like Alms. Haskell's type classes (Wadler and Blott 1989), a general mechanism invented to solve specific problems, have since found myriad unanticipated uses. Similarly, a practical, general form of substructural types as offered by Alms likely has many applications waiting to be uncovered.

We have now written several thousand lines of code in Alms, and this experience has led to improvements in both its design and our skill at using what it has to offer. For example, an earlier version of Alms had only unlimited ($\xrightarrow{\mathsf{U}}$) and affine ($\xrightarrow{\mathsf{A}}$) arrows, but Alms's behavioral contract library motivated the introduction of arrows whose kinds involve type variables (*e.g.,* $\xrightarrow{\langle\hat{\alpha}\rangle}$). In particular, we found ourselves writing the same function multiple times with different qualifiers in the argument types, and the addition of usage qualifier expressions eliminates this redundancy.

4. The Calculus ${}^a\lambda_{ms}$

We model Alms with a calculus based on System $\mathrm{F}^{\omega}_{<:}$, the higher-order polymorphic λ calculus with subtyping (Pierce 2002). Our calculus, ${}^a\lambda_{ms}$, makes several important changes to $\mathrm{F}^{\omega}_{<:}$:

- Our type system's structural rules are limited. In particular contraction, which duplicates variables in a typing context to make them accessible to multiple subterms, applies only to variables whose type is of the unlimited kind. Variables of affine or polymorphic kind cannot be contracted.
- Our kind system is enriched with dependent kinds.
- Our kind system is also enriched with variance on type operators (Steffen 1997), which allows abstract type constructors to specify how their results vary in relation to their parameters.
- Type operators are limited to first-order kinds—that is, type operators may not take type operators as parameters.
- Universally-quantified type variables are bounded only by a kind, not by a type.
- The subtyping relation is induced by the subtyping rule for functions, whereby an unlimited-use function may be supplied where a one-use function is expected.

The calculus ${}^a\lambda_{ms}$ also includes more types and terms than a minimal presentation of $\mathrm{F}^{\omega}_{<:}$. Because we are interested in practical issues, we think it is important that our model include products,[3] sums, mutable references, and non-termination.

We do not model modules directly in ${}^a\lambda_{ms}$, but its higher-kinded type abstraction makes ${}^a\lambda_{ms}$ a suitable target for the first-order fragment of Rossberg et al.'s (2010) "F-ing modules" translation.

[3] In linear logic terms, our calculus (like Alms) supplies multiplicative products ($\otimes$) and additive sums ($\oplus$) directly. Additive products ($\&$) are easily encoded by

$$\tau_1 \,\&\, \tau_2 \triangleq \forall\alpha{:}\mathsf{A}.\,(\tau_1 \xrightarrow{\mathsf{A}} \alpha) + (\tau_2 \xrightarrow{\mathsf{A}} \alpha) \xrightarrow{\varphi_1 \sqcup \varphi_2} \alpha$$

$$[e_1, e_2] \triangleq \Lambda\alpha{:}\mathsf{A}.\,\lambda k.\ \mathsf{case}\ k\ \mathsf{of}\ \iota_1\ x_1 \to x_1\, e_1;\ \iota_2\ x_2 \to x_2\, e_2,$$

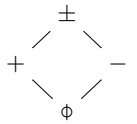
where φ_1 and φ_2 are the kinds of τ_1 and τ_2.

$$
\begin{array}{rcll}
\varphi & ::= & \mathsf{q} \mid \langle\alpha\rangle \mid \varphi_1 \sqcup \varphi_2 \mid \varphi_1 \sqcap \varphi_2 & \textit{(usage qualifiers)} \\
\kappa & ::= & \varphi \mid \Pi\alpha^{\mathfrak{v}}.\kappa & \textit{(kinds)} \\
\tau & ::= & \alpha \mid \lambda\alpha.\tau \mid \tau_1\,\tau_2 \mid \forall\alpha{:}\kappa.\tau \mid \tau_1 \xrightarrow{\varphi} \tau_2 & \textit{(types)} \\
 & \mid & 1 \mid (+) \mid (\times) \mid \mathsf{aref} & \\
e & ::= & x \mid \lambda x{:}\tau.e \mid e_1\,e_2 \mid \Lambda\alpha{:}\kappa.v \mid e[\tau] & \textit{(terms)} \\
 & \mid & \langle\rangle \mid \langle e_1, e_2\rangle \mid \mathsf{case}\ e_1\ \mathsf{of}\ \langle x_1, x_2\rangle \to e_2 & \\
 & \mid & \iota_1\ e \mid \iota_2\ e \mid \mathsf{case}\ e\ \mathsf{of}\ \iota_1\ x \to e_1;\ \iota_2\ y \to e_2 & \\
 & \mid & \mathsf{fix}\ e \mid \mathsf{new}\ e \mid \mathsf{swap}\ e_1\,e_2 \mid \mathsf{delete}\ e \mid \mathsf{ptr}\ \ell & \\
v & ::= & \lambda x{:}\tau.e \mid \Lambda\alpha{:}\kappa.v \mid \langle\rangle \mid \langle v_1, v_2\rangle & \textit{(values)} \\
 & \mid & \iota_1\ v \mid \iota_2\ v \mid \mathsf{ptr}\ \ell & \\
\Gamma, \Sigma & ::= & \cdot \mid \alpha{:}\kappa \mid x{:}\tau \mid \ell{:}\tau \mid \Gamma_1, \Gamma_2 & \textit{(typing contexts)}
\end{array}
$$

$(\mathfrak{q}, \sqsubseteq, \sqcup, \sqcap)$

(qual. constants)

$(\mathfrak{v}, \sqsubseteq, \sqcup, \sqcap)$



(variances)

$\mathfrak{v}_1 \cdot \mathfrak{v}_2 = \mathfrak{v}$

$\cdot$	±	−	+	ϕ
±	±	±	±	ϕ
−	±	+	−	ϕ
+	±	−	+	ϕ
ϕ	ϕ	ϕ	ϕ	ϕ

(variance composition)

Figure 3. Syntax of ${}^a\lambda_{ms}$

$s; e \longmapsto s'; e'$ (small-step reduction)

$$
\begin{array}{rcl}
s; (\lambda x{:}\tau.e)\ v & \longmapsto & s; [v/x]e \\
s; (\Lambda\alpha{:}\kappa.v)[\tau] & \longmapsto & s; [\tau/\alpha]v \\
s; \mathsf{fix}\ v & \longmapsto & s; v\ (\mathsf{fix}\ v) \\
s; \mathsf{new}\ v & \longmapsto & s \uplus \{\ell \mapsto v\}; \mathsf{ptr}\ \ell \\
s \uplus \{\ell \mapsto v_1\}; \mathsf{swap}\ (\mathsf{ptr}\ \ell)\ v_2 & \longmapsto & s \uplus \{\ell \mapsto v_2\}; \langle \mathsf{ptr}\ \ell, v_1\rangle \\
s \uplus \{\ell \mapsto v\}; \mathsf{delete}\ (\mathsf{ptr}\ \ell) & \longmapsto & s; \langle\rangle
\end{array}
$$

$$\frac{s; e \longmapsto s'; e'}{s; E[e] \longmapsto s'; E[e']}$$

$$
\begin{array}{rcll}
E & ::= & [] \mid E\,e \mid v\,E \mid E[\tau] \mid \langle E, e\rangle \mid \langle v, E\rangle & \textit{(evaluation} \\
 & \mid & \mathsf{case}\ E\ \mathsf{of}\ \langle x_1, x_2\rangle \to e \mid \iota_1\ E \mid \iota_2\ E & \textit{contexts)} \\
 & \mid & \mathsf{case}\ E\ \mathsf{of}\ \iota_1\ x \to e_1;\ \iota_2\ y \to e_2 & \\
 & \mid & \mathsf{fix}\ E \mid \mathsf{new}\ E \mid \mathsf{swap}\ E\,e \mid \mathsf{swap}\ v\,E \mid \mathsf{delete}\ E & \\
s & ::= & \{\} \mid \{\ell \mapsto v\} \mid s_1 \uplus s_2 & \textit{(stores)}
\end{array}
$$

Figure 4. Operational semantics (selected rules)

4.1 Syntax

We begin with the syntax of ${}^a\lambda_{ms}$ in figure 3. Terms (e) include the usual terms from System F (variables, abstractions, applications, type abstractions, and type applications), several forms for data construction and elimination (nil, pairs, pair elimination, sum injections, and sum elimination), recursion ($\mathsf{fix}\ e$), and several operations on reference cells (allocation, linear swap, and deallocation). Location names ($\mathsf{ptr}\ \ell$) appear at run time but are not present in source terms. Values (v) are standard.

Types (τ) include type variables, type-level abstraction and application, universal quantification, function types, and type constructor constants for unit, sums, products, and references. As in Alms, the function arrow carries a *usage qualifier* (φ), which specifies whether the function is unlimited or one-use.

The two constant usage qualifiers (q), U for unlimited and A for affine, are the bottom and top of the two-element lattice in figure 3. Now consider the K combinator $\Lambda\alpha{:}\langle\alpha\rangle.\,\Lambda\beta{:}\langle\beta\rangle.\,\lambda x{:}\alpha.\,\lambda y{:}\beta.\,x$ partially applied to a value: $\mathrm{K}[\tau_1][\tau_2]\ v$. Whether it is safe to duplicate this term depends on whether it is safe to duplicate v, and this is reflected in the instantiation of α. To express this relationship, we introduce usage qualifier expressions (φ), which form a bounded, distributive lattice over type variables with U and A as bottom and top. We can thus give K type $\forall\alpha{:}\langle\alpha\rangle.\,\forall\beta{:}\langle\beta\rangle.\,\alpha \xrightarrow{\mathsf{U}} \beta \xrightarrow{\langle\alpha\rangle} \alpha$.

Qualifier expressions are the base kinds of ${}^a\lambda_{ms}$—that is, the kinds that classify proper types that may in turn classify values. To classify type operators, kinds (κ) also include dependent product kinds, written $\Pi\alpha^{\mathfrak{v}}.\kappa$. This is the kind of a type operator that, when applied to a type with kind φ, gives a type with kind $[\varphi/\alpha]\kappa$. For example, the kind of the Alms list type constructor in §2 is $\Pi\alpha^{+}.\langle\alpha\rangle$, which means that $\mathsf{list}\ \tau$ has the same kind as τ.[4]

The superscript $+$ in kind $\Pi\alpha^{+}.\langle\alpha\rangle$ means that list is a covariant (or monotone) type constructor: if τ_1 is a subtype of τ_2 then $\mathsf{list}\ \tau_1$ is a subtype of $\mathsf{list}\ \tau_2$. Variances ($\mathfrak{v}$) form a four-point lattice (figure 3). A type operator may also be contravariant ($-$), where the result varies inversely with the argument; omnivariant (ϕ), where argument may vary freely without affecting the result; or invariant ($\pm$) where the argument may not vary at all without producing a subtyping-unrelated result. We define a composition operation ($\cdot$) on variances, where $\mathfrak{v}_1 \cdot \mathfrak{v}_2$ is the variance of the composition of type operators having variances $\mathfrak{v}_1$ and $\mathfrak{v}_2$.

The kinds of the type constructors for sums and references may aid understanding. The sum type constructor $(+)$ has kind $\Pi\alpha^{+}.\,\Pi\beta^{+}.\,\langle\alpha\rangle \sqcup \langle\beta\rangle$. This means that the kind of a sum type is at least as restrictive as the kinds of its disjuncts. It is covariant in both arguments, which means that $\tau_1 + \tau_2$ is a subtype of $\tau_1' + \tau_2'$ if τ_1 is a subtype of τ_1' and τ_2 is a subtype of τ_2'. The reference type constructor, on the other hand, has kind $\Pi\alpha^{\pm}.\,\mathsf{A}$. This means that reference cells are always affine and that their types do not support subtyping in either direction.

Typing contexts (Γ or Σ) associate type variables with their kinds, variables with their types, and locations with the types of their contents. By convention, we use Γ for typing contexts that include neither affine variables nor locations, and we use Σ for typing contexts that may include locations and affine (or indeterminate) variables. We use $\mathrm{dom}(\Gamma)$ to refer to the type variables, variables, and locations mapped by a context.

We define the free type variables in a variety of syntactic forms ($\mathrm{ftv}(e)$, $\mathrm{ftv}(\tau)$, $\mathrm{ftv}(\kappa)$, etc.) in the standard way. We use $\mathrm{locs}(e)$ to denote the set of location names present in term e. There are no binders for locations.

4.2 Operational Semantics

The operational semantics of ${}^a\lambda_{ms}$ is mostly a standard call-by-value reduction semantics. We give a selection of rules in figure 4. The reduction relation ($\longmapsto$) relates configurations ($s; e$) comprising a store and a term. A store maps locations (ℓ) to values (v). Stores are taken to be unordered and do not repeat location names.

The rules for reference operations are worth noting. In store s, $\mathsf{new}\ v$ chooses a fresh location ℓ, adding v to the store at location ℓ and reducing to the reference $\mathsf{ptr}\ \ell$. The operation $\mathsf{swap}\ (\mathsf{ptr}\ \ell)\ v_2$ requires that the store have location ℓ holding some value v_1. It swaps v_2 for v_1 in the store, returning a pair of a reference to ℓ and value v_1. Finally, $\mathsf{delete}\ (\mathsf{ptr}\ \ell)$ also requires that the store contain ℓ, which it then removes from the store. This means that freeing a location can result in a dangling pointer, which would cause subsequent attempts to access that location to get stuck. Our type system prevents this.

[4] Whereas Alms uses ML's conventional postfix notation for type-level application, ${}^a\lambda_{ms}$ uses prefix application.

$\Gamma \vdash \kappa$	kind κ is well formed	(fig. 6)
$\Gamma \vdash \kappa_1 <: \kappa_2$	kind κ_1 is subsumed by κ_2	(fig. 6)
$\Gamma \vdash \alpha \in \tau \updownarrow \mathfrak{v}$	type τ varies $\mathfrak{v}$-ly when α increases	(fig. 7)
$\Gamma \vdash \tau : \kappa$	type τ has kind κ	(fig. 7)
$\Gamma \vdash \tau_1 <:^{\mathfrak{v}} \tau_2$	type τ_1 is $\mathfrak{v}$-related to type τ_2	(fig. 7)
$\Gamma \vdash \Sigma \preceq \varphi$	context Σ is bounded by qualifier φ	(fig. 8)
$\vdash (\Gamma_0; \Sigma_0), \Sigma' \rightsquigarrow \Gamma; \Sigma$	extending $\Gamma_0; \Sigma_0$ with Σ' gives $\Gamma; \Sigma$	(fig. 8)
$\Gamma; \Sigma \rhd e : \tau$	term e has type τ	(fig. 9)
$\Sigma_1 \rhd s : \Sigma_2$	store s has type Σ_2	(fig. 10)
$\rhd s; e : \tau$	configuration $s; e$ has type τ	(fig. 10)

Figure 5. Type system judgments

$\boxed{\Gamma \vdash \kappa}$ (kind well-formedness)

$$\textsc{OK-Qual}\ \frac{\mathrm{ftv}(\varphi) \subseteq \mathrm{dom}(\Gamma)}{\Gamma \vdash \varphi} \qquad \textsc{OK-Arr}\ \frac{\Gamma, \alpha{:}\langle\alpha\rangle \vdash \kappa \quad \text{if } \alpha \in \mathrm{ftv}(\kappa) \text{ then } + \sqsubseteq \mathfrak{v}}{\Gamma \vdash \Pi\alpha^{\mathfrak{v}}.\kappa}$$

$\boxed{\Gamma \vdash \kappa_1 <: \kappa_2}$ (subkinding)

$$\textsc{KSub-Qual}\ \frac{\Gamma \models \varphi_1 \sqsubseteq \varphi_2}{\Gamma \vdash \varphi_1 <: \varphi_2} \qquad \textsc{KSub-Arr}\ \frac{\mathfrak{v}_1 \sqsubseteq \mathfrak{v}_2 \qquad \Gamma, \alpha{:}\langle\alpha\rangle \vdash \kappa_1 <: \kappa_2}{\Gamma \vdash \Pi\alpha^{\mathfrak{v}_1}.\kappa_1 <: \Pi\alpha^{\mathfrak{v}_2}.\kappa_2}$$

Figure 6. Statics (i): kinds

4.3 Static Semantics

Our type system involves a large number of judgments, which we summarize in figure 5. In several cases, we omit premises concerning well-formedness of contexts, which clutter the presentation and do not add anything of interest. The omitted premises appear in the extended version of this paper.

Kind judgments. Judgments on kinds appear in figure 6. The first judgment, $\Gamma \vdash \kappa$, determines whether a kind κ is well formed in typing context Γ. A base kind (*i.e.,* a usage qualifier expression) is well formed provided that Γ specifies a kind for all of its free variables. A dependent product kind $\Pi\alpha^{\mathfrak{v}}.\kappa$ is well formed if Γ maps all of its free variables, provided it satisfies a second condition: whenever the bound type variable α is free in κ—that is, when the kind is truly dependent—then variance $\mathfrak{v}$ must be $+$ or $\pm$. This rules out incoherent kinds such as $\Pi\alpha^{-}.\langle\alpha\rangle$ that classify no useful type operator but whose presence breaks the kinding relation's monotonicity property (see lemma 4).

The second judgment is subkinding: $\Gamma \vdash \kappa_1 <: \kappa_2$. As we will see, if a type has kind κ_1, then it may be used where κ_1 or any greater kind is expected. For dependent product kinds the subkinding order is merely the product order on the variance and the result kind, but for base kinds the relation relies on an interpretation of qualifier expressions.

We interpret qualifier expressions via a valuation $\mathcal{V}$, which is a map from type variables to qualifier constants. We extend $\mathcal{V}$'s domain to qualifier expressions:

$$\mathcal{V}(\mathfrak{q}) = \mathfrak{q} \qquad \mathcal{V}(\varphi_1 \sqcup \varphi_2) = \mathcal{V}(\varphi_1) \sqcup \mathcal{V}(\varphi_2)$$
$$\mathcal{V}(\langle\alpha\rangle) = \mathcal{V}(\alpha) \qquad \mathcal{V}(\varphi_1 \sqcap \varphi_2) = \mathcal{V}(\varphi_1) \sqcap \mathcal{V}(\varphi_2)$$

We need to interpret qualifier expressions under a typing context:

Definition 1 (Consistent valuations)**.** *A valuation $\mathcal{V}$ is **consistent with** a typing context Γ if for all $\alpha{:}\varphi \in \Gamma$, $\mathcal{V}(\alpha) \sqsubseteq \mathcal{V}(\varphi)$.*

Thus, a valuation is consistent with a context if it corresponds to a potential instantiation of the type variables, given that context.

Definition 2 (Qualifier subsumption)**.** *We say that φ_1 is **subsumed by** φ_2 in Γ, written $\Gamma \models \varphi_1 \sqsubseteq \varphi_2$, if for all valuations $\mathcal{V}$ consistent with Γ, $\mathcal{V}(\varphi_1) \sqsubseteq \mathcal{V}(\varphi_2)$.*

In other words, in all possible instantiations of the type variables in Γ, qualifier φ_1 being A implies that φ_2 is A.

Kinding and variance. The first two judgments in figure 7, for computing variances and giving kinds to types, are defined by mutual induction. It should be clear on inspection that the definitions are well-founded. Judgment $\Gamma \vdash \alpha \in \tau \updownarrow \mathfrak{v}$ means that type variable α appears in type τ at variance $\mathfrak{v}$, or in other words, that type operator $\lambda\alpha.\tau$ has variance $\mathfrak{v}$. Rules V-Var and V-Bot say that type variables appear positively with respect to themselves and omnivariantly with respect to types in which they are not free. Rule V-Abs says that a type variable appears in a type operator $\lambda\beta.\tau$ at the same variance that it appears in the body τ. The remaining three rules are more involved:

- By rule V-App, the variance of a type variable in a type application comes from both the operator and the operand. The variance of α in $\tau_1\,\tau_2$ is at least the variance of α in τ_1 and at least the variance of α in the τ_2 composed with the variance of operator τ_1. This makes sense: if τ is a contravariant type operator, then α appears negatively in $\tau\,\alpha$ but positively in $\tau\,(\tau\,\alpha)$.
- By rule V-All, the variance of α in $\forall\beta{:}\kappa.\tau$ is at least its variance in τ. However, if α appears in κ then it is invariant in $\forall\beta{:}\kappa.\tau$. This reflects the fact that universally-quantified types are related only if their bounds (κ) match exactly, so changing a type variable that appears in κ produces an unrelated type. (This means that ${}^{a}\lambda_{ms}$ is based on the *kernel* variant of $\mathrm{F}^{\omega}_{<:}$ (Pierce 2002).)
- By rule V-Arr, the variance of α in a function type $\tau_1 \xrightarrow{\varphi} \tau_2$ is at least its variance in the codomain τ_2 and at least the opposite (composition with $-$) of its variance in the domain τ_1. This reflects function argument contravariance. The variance of α is at least $+$ if it appears in the qualifier expression φ.

The second judgment, $\Gamma \vdash \tau : \kappa$, assigns kinds to well formed types. Rule K-Var merely looks up the kind of a type variable in the context. Rules K-Abs and K-App are the usual rules for dependent abstraction and application, with two small changes in rule K-Abs. First, it associates α with *itself* in the context, as $\alpha{:}\langle\alpha\rangle$, which ensures that occurrences of α in τ can be reflected in κ. Second, it appeals to the variance judgment to determine the variance of the type operator. Rule K-All assigns a universal type the same kind as its body, but with A replacing α. This is necessary because the resulting kind is outside the scope of α. Qualifier A is a safe bound for any instantiation of α, and no terms have types that lose precision by this choice. The kind of an arrow type, in rule K-Arr, is just the qualifier expression attached to the arrow. The remaining rules give kinds for type constructor constants.

Type equivalence and dereliction subtyping. The last judgment in figure 7 is subtyping. The subtyping relation is parametrized by a variance $\mathfrak{v}$, which gives the direction of the subtyping: $\Gamma \vdash \tau_1 <:^{+} \tau_2$ is the usual direction, judging τ_1 a subtype of τ_2. In terms of subsumption, this means that values of type τ_1 may be used where values of type τ_2 are expected. The other variances are useful in defining the relation in the presence of $\mathfrak{v}$-variant type operators: $(<:^{-})$ gives the inverse of the subtyping relation, $(<:^{\pm})$ relates only equivalent types, and $(<:^{\Phi})$ relates all types. We can see how this works this in rule TSub-App. To determine whether $\tau_{11}\,\tau_{12}$ is a

$\boxed{\Gamma \vdash \alpha \in \tau \updownarrow \mathfrak{v}}$ (variance of type variables with respect to types)

$$\textsc{V-Var}\ \frac{}{\Gamma \vdash \alpha \in \alpha \updownarrow +} \qquad \textsc{V-Bot}\ \frac{\alpha \notin \mathrm{ftv}(\tau)}{\Gamma \vdash \alpha \in \tau \updownarrow \phi} \qquad \textsc{V-Abs}\ \frac{\Gamma, \beta{:}\langle\beta\rangle \vdash \alpha \in \tau \updownarrow \mathfrak{v}}{\Gamma \vdash \alpha \in \lambda\beta.\tau \updownarrow \mathfrak{v}}$$

$$\textsc{V-App}\ \frac{\Gamma \vdash \alpha \in \tau_1 \updownarrow \mathfrak{v}_1 \qquad \Gamma \vdash \alpha \in \tau_2 \updownarrow \mathfrak{v}_2 \qquad \Gamma \vdash \tau_1 : \Pi\beta^{\mathfrak{v}_3}.\kappa_3}{\Gamma \vdash \alpha \in \tau_1\,\tau_2 \updownarrow \mathfrak{v}_1 \sqcup (\mathfrak{v}_2 \cdot \mathfrak{v}_3)} \qquad \textsc{V-All}\ \frac{\Gamma, \beta{:}\kappa \vdash \alpha \in \tau \updownarrow \mathfrak{v}_1 \qquad \mathfrak{v}_2 = \text{if } \alpha \in \mathrm{ftv}(\kappa) \text{ then } \pm \text{ else } \phi}{\Gamma \vdash \alpha \in \forall\beta{:}\kappa.\tau \updownarrow \mathfrak{v}_1 \sqcup \mathfrak{v}_2}$$

$$\textsc{V-Arr}\ \frac{\Gamma \vdash \alpha \in \tau_1 \updownarrow \mathfrak{v}_1 \qquad \Gamma \vdash \alpha \in \tau_2 \updownarrow \mathfrak{v}_2 \qquad \mathfrak{v}_3 = \text{if } \alpha \in \mathrm{ftv}(\varphi) \text{ then } + \text{ else } \phi}{\Gamma \vdash \alpha \in \tau_1 \xrightarrow{\varphi} \tau_2 \updownarrow -\mathfrak{v}_1 \sqcup \mathfrak{v}_2 \sqcup \mathfrak{v}_3}$$

$\boxed{\Gamma \vdash \tau : \kappa}$ (kinding of types)

$$\textsc{K-Var}\ \frac{\alpha{:}\kappa \in \Gamma}{\Gamma \vdash \alpha{:}\kappa} \qquad \textsc{K-Abs}\ \frac{\Gamma, \alpha{:}\langle\alpha\rangle \vdash \tau : \kappa \qquad \Gamma, \alpha{:}\langle\alpha\rangle \vdash \alpha \in \tau \updownarrow \mathfrak{v}}{\Gamma \vdash \lambda\alpha.\tau : \Pi\alpha^{\mathfrak{v}}.\kappa} \qquad \textsc{K-App}\ \frac{\Gamma \vdash \tau_1 : \Pi\alpha^{\mathfrak{v}}.\kappa \qquad \Gamma \vdash \tau_2 : \varphi}{\Gamma \vdash \tau_1\,\tau_2 : [\varphi/\alpha]\kappa}$$

$$\textsc{K-All}\ \frac{\Gamma, \alpha{:}\kappa \vdash \tau : \varphi}{\Gamma \vdash \forall\alpha{:}\kappa.\tau : [\mathsf{A}/\alpha]\varphi} \qquad \textsc{K-Arr}\ \frac{\Gamma \vdash \tau_1 : \varphi_1 \qquad \Gamma \vdash \tau_2 : \varphi_2}{\Gamma \vdash \tau_1 \xrightarrow{\varphi} \tau_2 : \varphi} \qquad \textsc{K-Unit}\ \frac{}{\Gamma \vdash 1 : \mathsf{U}}$$

$$\textsc{K-Sum}\ \frac{}{\Gamma \vdash (+) : \Pi\alpha^{+}.\Pi\beta^{+}.\langle\alpha\rangle \sqcup \langle\beta\rangle} \qquad \textsc{K-Prod}\ \frac{}{\Gamma \vdash (\times) : \Pi\alpha^{+}.\Pi\beta^{+}.\langle\alpha\rangle \sqcup \langle\beta\rangle} \qquad \textsc{K-Ref}\ \frac{}{\Gamma \vdash \mathsf{aref} : \Pi\alpha^{\pm}.\mathsf{A}}$$

$\boxed{\Gamma \vdash \tau_1 <:^{\mathfrak{v}} \tau_2}$ (subtyping)

$$\textsc{TSub-Eq}\ \frac{\tau_1 \equiv \tau_2}{\Gamma \vdash \tau_1 <:^{\mathfrak{v}} \tau_2} \qquad \textsc{TSub-Trans}\ \frac{\Gamma \vdash \tau_1 <:^{\mathfrak{v}} \tau_2 \qquad \Gamma \vdash \tau_2 <:^{\mathfrak{v}} \tau_3}{\Gamma \vdash \tau_1 <:^{\mathfrak{v}} \tau_3} \qquad \textsc{TSub-Omni}\ \frac{}{\Gamma \vdash \tau_1 <:^{\phi} \tau_2}$$

$$\textsc{TSub-Contra}\ \frac{\Gamma \vdash \tau_2 <:^{-\mathfrak{v}} \tau_1}{\Gamma \vdash \tau_1 <:^{\mathfrak{v}} \tau_2} \qquad \textsc{TSub-Abs}\ \frac{\Gamma, \alpha{:}\langle\alpha\rangle \vdash \tau_1 <:^{\mathfrak{v}} \tau_2}{\Gamma \vdash \lambda\alpha.\tau_1 <:^{\mathfrak{v}} \lambda\alpha.\tau_2}$$

$$\textsc{TSub-App}\ \frac{\Gamma \vdash \tau_{11} : \Pi\alpha^{\mathfrak{v}_1}.\kappa_1 \qquad \Gamma \vdash \tau_{21} : \Pi\alpha^{\mathfrak{v}_2}.\kappa_2 \qquad \Gamma \vdash \tau_{11} <:^{\mathfrak{v}} \tau_{21} \qquad \Gamma \vdash \tau_{12} <:^{\mathfrak{v}\cdot(\mathfrak{v}_1 \sqcup \mathfrak{v}_2)} \tau_{22}}{\Gamma \vdash \tau_{11}\,\tau_{12} <:^{\mathfrak{v}} \tau_{21}\,\tau_{22}}$$

$$\textsc{TSub-All}\ \frac{\Gamma, \alpha{:}\kappa \vdash \tau_1 <:^{\mathfrak{v}} \tau_2}{\Gamma \vdash \forall\alpha{:}\kappa.\tau_1 <:^{\mathfrak{v}} \forall\alpha{:}\kappa.\tau_2}$$

$$\textsc{TSub-Arr}\ \frac{\Gamma \vdash \tau_{11} <:^{-\mathfrak{v}} \tau_{21} \qquad \Gamma \vdash \tau_{12} <:^{\mathfrak{v}} \tau_{22} \qquad \Gamma \vdash \varphi_1 <:^{\mathfrak{v}} \varphi_2}{\Gamma \vdash \tau_{11} \xrightarrow{\varphi_1} \tau_{12} <:^{\mathfrak{v}} \tau_{21} \xrightarrow{\varphi_2} \tau_{22}}$$

Figure 7. Statics (ii): types

subtype of $\tau_{21}\,\tau_{22}$, we take $\mathfrak{v}$ to be $+$, yielding

$$\frac{\Gamma \vdash \tau_{11} : \Pi\alpha^{\mathfrak{v}_1}.\kappa_1 \qquad \Gamma \vdash \tau_{21} : \Pi\alpha^{\mathfrak{v}_2}.\kappa_2 \qquad \Gamma \vdash \tau_{11} <:^{+} \tau_{21} \qquad \Gamma \vdash \tau_{12} <:^{\mathfrak{v}_1 \sqcup \mathfrak{v}_2} \tau_{22}}{\Gamma \vdash \tau_{11}\,\tau_{12} <:^{+} \tau_{21}\,\tau_{22}}.$$

This means that for the subtyping relation to hold:

- The operators must be related in the same direction, so that τ_{11} is a subtype of τ_{21}.
- The operands must be related in the direction given by the variances of the operators. For example, if both operators are covariant, then the operands must vary in the same direction, so that τ_{12} is a subtype of τ_{22}. If both operators are contravariant, then the operands must vary in the opposite direction. If the operators are invariant then the operands cannot vary at all, but if they are omnivariant then $\tau_{11}\,\tau'_{12}$ is a subtype of $\tau_{21}\,\tau'_{22}$ for any τ'_{12} and τ'_{22}.

Rule TSub-Eq says that subtyping includes type equivalence ($\tau_1 \equiv \tau_2$), which is merely β equivalence on types. Rule TSub-Omni allows any pair of types to be related by ϕ-variant subtyping, and rule TSub-Contra says that the opposite variance sign gives the inverse relation. Rules TSub-Abs and TSub-All specify that type operators and universally-quantified types are related if their bodies are.

Rule TSub-Arr is more than the usual arrow subtyping rule. Beyond the usual contravariance for arguments and covariance for results, it requires that qualifiers φ_1 and φ_2 relate in the same direction. This rule is the source of non-trivial subtyping in ${}^{a}\lambda_{ms}$, without which subtyping would relate only equivalent types. The rule has two important implications.

First, an unlimited-use function can always be used where a one-use function is expected. This corresponds to linear logic's usual dereliction rule, which says that the ! ("of course!") modality may always be removed. ILL (Bierman 1993) has a rule:

$$\frac{\Delta \vdash e : !A}{\Delta \vdash \mathsf{derelict}\ e : A}\ \textsc{Dereliction}.$$

Dereliction is syntax-directed in this rule, but for practical programming we consider that as too inconvenient. Thus, our subtyping relation supports dereliction as needed.

For example, the function for creating a new thread in Alms, *Thread.fork*, has type $\forall\hat{\alpha}.\,(\mathsf{unit} \xrightarrow{\mathsf{A}} \hat{\alpha}) \xrightarrow{\mathsf{U}} \hat{\alpha}\,\mathsf{thread}$, which means that *Thread.fork* will not call its argument more than once. However, this should not stop us from passing an unlimited-use function to *Thread.fork*, and indeed we can. Dereliction subtyping allows us to use a value of type $\mathsf{unit} \xrightarrow{\mathsf{U}} \hat{\alpha}$ where a value of type $\mathsf{unit} \xrightarrow{\mathsf{A}} \hat{\alpha}$ is expected. Alternatively, by domain contravariance, we can use *Thread.fork* where a value of type $\forall\hat{\alpha}.\,(\mathsf{unit} \xrightarrow{\mathsf{U}} \hat{\alpha}) \xrightarrow{\mathsf{U}} \hat{\alpha}\,\mathsf{thread}$ is expected. In this case subsumption allows us to forget *Thread.fork*'s promise not to reuse its argument.

The other important implication of dereliction subtyping will become clearer once we see how qualifier expressions are assigned to function types. Subsumption makes it reasonable to always assign functions the most permissive safe usage qualifier, because

$\boxed{\Gamma \vdash \Sigma \preceq \varphi}$ (bound of typing context)

$$\textsc{B-Nil}\ \frac{}{\Gamma \vdash \cdot \preceq \mathsf{U}} \qquad \textsc{B-ConsA}\ \frac{\Gamma \vdash \Sigma \preceq \varphi_1 \qquad \Gamma \vdash \kappa}{\Gamma \vdash \Sigma, \alpha{:}\kappa \preceq \varphi_1}$$

$$\textsc{B-ConsX}\ \frac{\Gamma \vdash \Sigma \preceq \varphi_1 \qquad \Gamma \vdash \tau : \varphi_2}{\Gamma \vdash \Sigma, x{:}\tau \preceq \varphi_1 \sqcup \varphi_2} \qquad \textsc{B-ConsL}\ \frac{\Gamma \vdash \Sigma \preceq \varphi_1 \qquad \Gamma \vdash \tau : \varphi_2}{\Gamma \vdash \Sigma, \ell{:}\tau \preceq \mathsf{A}}$$

$\boxed{\vdash (\Gamma_0; \Sigma_0), \Sigma' \rightsquigarrow \Gamma_1; \Sigma_1}$ (typing context extension)

$$\textsc{X-Nil}\ \frac{}{\vdash (\Gamma; \Sigma), \cdot \rightsquigarrow \Gamma; \Sigma}$$

$$\textsc{X-ConsU}\ \frac{\Gamma_0 \vdash \tau : \mathsf{U} \qquad \vdash (\Gamma_0, x{:}\tau; \Sigma_0), \Sigma' \rightsquigarrow \Gamma; \Sigma}{\vdash (\Gamma_0; \Sigma_0), x{:}\tau, \Sigma' \rightsquigarrow \Gamma; \Sigma} \qquad \textsc{X-ConsA}\ \frac{\Gamma_0 \vdash \tau : \varphi \qquad \vdash (\Gamma_0; \Sigma_0, x{:}\tau), \Sigma' \rightsquigarrow \Gamma; \Sigma}{\vdash (\Gamma_0; \Sigma_0), x{:}\tau, \Sigma' \rightsquigarrow \Gamma; \Sigma}$$

Figure 8. Statics (iii): typing contexts

subsumption then allows us to use them in a less permissive context. Dereliction subtyping applies only to function types because in both the ${}^a\lambda_{ms}$ calculus and Alms language only function types carry qualifiers. For instance, Alms has no separate types $\mathsf{int}^{\mathsf{U}}$ for unlimited integers and $\mathsf{int}^{\mathsf{A}}$ for affine integers. Integers are always unlimited. If a programmer wants an affine version of int, she can create it in Alms using the module system.

Context judgments. Figure 8 defines two judgments on contexts. Judgment $\Gamma \vdash \Sigma \preceq \varphi$, which will be important in typing functions, computes an upper bound φ on the qualifiers of all the types in context Σ. If a context contains any locations, it is bounded by A; otherwise, its bound is the least upper bound of the qualifiers of all the types of variables in the context.

The second judgment shows how environments are extended by variable bindings. The typing judgment for terms will use two typing contexts: Γ holds environment information that may be safely duplicated, such as type variables and variables of unlimited type, whereas Σ holds information, such as location types and affine variables, that disallows duplication. Given contexts Γ_0 and Σ_0, judgment $\vdash (\Gamma_0; \Sigma_0), \Sigma' \rightsquigarrow \Gamma; \Sigma$ extends them by the variables and types in Σ' to get Γ and Σ. Any variables may be placed in Σ, but only variables whose types are known to be unlimited may be placed in Γ, since Γ may be duplicated.

Term judgment. The typing judgment for terms appears in figure 9. The judgment, $\Gamma; \Sigma \rhd e : \tau$, uses two typing contexts in the style of DILL (Barber 1996): the unlimited environment Γ and the affine environment Σ. When typing multiplicative terms such as application, we distribute Γ to both subterms but partition Σ between the two:

$$\frac{\Gamma; \Sigma_1 \rhd e_1 : \tau_1 \xrightarrow{\varphi} \tau_2 \qquad \Gamma; \Sigma_2 \rhd e_2 : \tau_1}{\Gamma; \Sigma_1, \Sigma_2 \rhd e_1\, e_2 : \tau_2}\ \textsc{T-App}$$

Unlike DILL, not all types in Σ are necessarily affine. Since types whose usage qualifier involves type variables are not known to be unlimited, we place those in Σ, to ensure that we do not duplicate values that *might* turn out to be affine once universally-quantified types are instantiated.

The other multiplicative rules are T-Pair for product introduction, T-Unpair for product elimination, and T-Swap for reference updates. Note that T-Swap does not require that the type of the reference in its first parameter match the type of the value in its second—in other words, swap performs a *strong update*. To type the term case e of $\langle x_1, x_2\rangle \to e_1$, rule T-Unpair first splits the affine environment into Σ_1 for typing subterm e and Σ_2 for subterm e_1. It invokes the context extension relation (figure 8) to extend Γ and Σ_2 with bindings for x_1 and x_2 in order to type e_1. The context extension relation requires that variables not known to be unlimited be added to Σ_2.

The rule for sum elimination, T-Choose, is both multiplicative and additive: the affine context is split between the term being destructed and the branches of the case expression. However, the portion of the context given to the branches is shared between them, because only one or the other will be evaluated. Rule T-Choose also uses the context extension relation to bind the pattern variables for the branches.

Rules T-New and T-Delete introduce and eliminate reference types in the usual way. Likewise, the sum introduction rules T-Ink and type abstraction rule T-TAbs are standard. Rules T-Var, T-Ptr, and T-Unit are standard for an affine calculus but not a linear one, as they implicitly support weakening by allowing Σ to contain unused bindings. Rule T-Fix is also standard, modulo the reasonable constraint that its parameter function be unlimited, since the reduction rule for fix makes a copy of the parameter.

The type application rule T-TApp supports subkinding, because it requires only that the kind of the actual type parameter be a subkind of that of the formal parameter. This is the rule that supports the sort of type abstraction that we used in our examples of §2 to construct affine capabilities. For example, the rule lets us instantiate affine type variable α with unlimited unit type 1:

$$\frac{\Gamma; \Sigma \rhd (\Lambda\alpha{:}\mathsf{A}.\, \lambda x{:}\alpha.\, e) : \forall\alpha{:}\mathsf{A}.\, \alpha \xrightarrow{\varphi} \tau \qquad \Gamma \vdash 1 : \mathsf{U} \qquad \Gamma \vdash \mathsf{U} <: \mathsf{A}}{\Gamma; \Sigma \rhd (\Lambda\alpha{:}\mathsf{A}.\, \lambda x{:}\alpha.\, e)[1] : 1 \xrightarrow{\varphi} \tau}\ \textsc{T-TApp}$$

Within its scope, α is considered *a priori* affine, regardless of how it may eventually be instantiated. This term types only if x appears in affine fashion in e.

This brings us finally to T-Abs, the rule for typing term-level λ abstractions. To type a term $\lambda x{:}\tau_1.\, e$, rule T-Abs uses the context extension relation to add $x{:}\tau_1$ to its contexts and types the body e in the extended contexts. It also must determine the qualifier φ that decorates the arrow. Because abstractions close over their free variables, duplicating a function also duplicates the values of its free variables. Therefore, the qualifier of a function type should be at least as restrictive as the qualifiers of the abstraction's free variables. To do this, rule T-Abs appeals to the context bounding judgment (figure 8) to find the least upper bound of the usage qualifiers of variables in the affine environment, and it requires that the function type's qualifier be equally restrictive.

This refines linear logic's usual promotion rule, which says that the ! modality may be added to propositions that in turn depend only on !-ed resources. In ILL, we have

$$\frac{!\Delta \vdash e : A}{!\Delta \vdash \mathsf{promote}\ e : !A}\ \textsc{Promotion},$$

where $!\Delta$ is a context in which all assumptions are of the form $x : !B$. As with dereliction, in our system it only makes sense to apply promotion to function types.

Our treatment of promotion indicates why we need the explicit weakening rule T-Weak, which allows discarding unused portions of the affine environment. In order to give a function type the best qualifier possible, we need to remove from Σ any unused variables or locations that might otherwise raise the bound on Σ, and the algorithmic version of the type system as implemented in

$\boxed{\Gamma; \Sigma \rhd e : \tau}$ (typing of terms)

$$\textsc{T-Subsume}\ \frac{\Gamma; \Sigma \rhd e : \tau' \qquad \Gamma \vdash \tau' <:^{+} \tau}{\Gamma; \Sigma \rhd e : \tau} \qquad \textsc{T-Weak}\ \frac{\Gamma; \Sigma \rhd e : \tau}{\Gamma; \Sigma, \Sigma' \rhd e : \tau} \qquad \textsc{T-Var}\ \frac{x{:}\tau \in \Gamma, \Sigma}{\Gamma; \Sigma \rhd x : \tau} \qquad \textsc{T-Ptr}\ \frac{\ell{:}\tau \in \Sigma}{\Gamma; \Sigma \rhd \mathsf{ptr}\ \ell : \mathsf{aref}\ \tau}$$

$$\textsc{T-Abs}\ \frac{\vdash (\Gamma; \Sigma), x{:}\tau_1 \rightsquigarrow \Gamma'; \Sigma' \qquad \Gamma'; \Sigma' \rhd e : \tau_2 \qquad \Gamma \vdash \Sigma \preceq \varphi}{\Gamma; \Sigma \rhd \lambda x{:}\tau_1.e : \tau_1 \xrightarrow{\varphi} \tau_2} \qquad \textsc{T-App}\ \frac{\Gamma; \Sigma_1 \rhd e_1 : \tau_1 \xrightarrow{\varphi} \tau_2 \qquad \Gamma; \Sigma_2 \rhd e_2 : \tau_1}{\Gamma; \Sigma_1, \Sigma_2 \rhd e_1\, e_2 : \tau_2} \qquad \textsc{T-TAbs}\ \frac{\Gamma, \alpha{:}\kappa; \Sigma \rhd v : \tau}{\Gamma; \Sigma \rhd \Lambda\alpha{:}\kappa.v : \forall\alpha{:}\kappa.\tau}$$

$$\textsc{T-TApp}\ \frac{\Gamma; \Sigma \rhd e : \forall\alpha{:}\kappa.\tau \qquad \Gamma \vdash \tau' : \kappa' \qquad \Gamma \vdash \kappa' <: \kappa}{\Gamma; \Sigma \rhd e[\tau'] : [\tau'/\alpha]\tau} \qquad \textsc{T-Unit}\ \frac{}{\Gamma; \Sigma \rhd \langle\rangle : 1} \qquad \textsc{T-Pair}\ \frac{\Gamma; \Sigma_1 \rhd e_1 : \tau_1 \qquad \Gamma; \Sigma_2 \rhd e_2 : \tau_2}{\Gamma; \Sigma_1, \Sigma_2 \rhd \langle e_1, e_2\rangle : \tau_1 \times \tau_2}$$

$$\textsc{T-Unpair}\ \frac{\Gamma; \Sigma_1 \rhd e : \tau_1 \times \tau_2 \qquad \vdash (\Gamma; \Sigma_2), x_1{:}\tau_1, x_2{:}\tau_2 \rightsquigarrow \Gamma'; \Sigma' \qquad \Gamma'; \Sigma' \rhd e_1 : \tau}{\Gamma; \Sigma_1, \Sigma_2 \rhd \mathsf{case}\ e\ \mathsf{of}\ \langle x_1, x_2\rangle \to e_1 : \tau} \qquad \textsc{T-In}k\ \frac{\Gamma; \Sigma \rhd e : \tau_k}{\Gamma; \Sigma \rhd \iota_k\ e : \tau_1 + \tau_2}\ {}^{(k \in \{1,2\})}$$

$$\textsc{T-Choose}\ \frac{\Gamma; \Sigma \rhd e : \tau_1 + \tau_2 \qquad \vdash (\Gamma; \Sigma'), x_1{:}\tau_1 \rightsquigarrow \Gamma_1; \Sigma_1 \qquad \Gamma_1; \Sigma_1 \rhd e_1 : \tau \qquad \vdash (\Gamma; \Sigma'), x_2{:}\tau_2 \rightsquigarrow \Gamma_2; \Sigma_2 \qquad \Gamma_2; \Sigma_2 \rhd e_2 : \tau}{\Gamma; \Sigma, \Sigma' \rhd \mathsf{case}\ e\ \mathsf{of}\ \iota_1\ x_1 \to e_1;\ \iota_2\ x_2 \to e_2 : \tau}$$

$$\textsc{T-Fix}\ \frac{\Gamma; \Sigma \rhd e : \tau \xrightarrow{\mathsf{U}} \tau}{\Gamma; \Sigma \rhd \mathsf{fix}\ e : \tau} \qquad \textsc{T-New}\ \frac{\Gamma; \Sigma \rhd e : \tau}{\Gamma; \Sigma \rhd \mathsf{new}\ e : \mathsf{aref}\ \tau} \qquad \textsc{T-Swap}\ \frac{\Gamma; \Sigma_1 \rhd e_1 : \mathsf{aref}\ \tau_1 \qquad \Gamma; \Sigma_2 \rhd e_2 : \tau_2}{\Gamma; \Sigma_1, \Sigma_2 \rhd \mathsf{swap}\ e_1\, e_2 : \mathsf{aref}\ \tau_2 \times \tau_1} \qquad \textsc{T-Delete}\ \frac{\Gamma; \Sigma \rhd e : \mathsf{aref}\ \tau}{\Gamma; \Sigma \rhd \mathsf{delete}\ e : 1}$$

Figure 9. Statics (iv): terms

$\boxed{\Sigma_1 \rhd s : \Sigma_2}$ (store typing)

$$\textsc{S-Nil}\ \frac{}{\Sigma_1 \rhd \{\} : \cdot} \qquad \textsc{S-Cons}\ \frac{\Sigma_{11} \rhd s : \Sigma_2 \qquad \cdot; \Sigma_{12} \rhd v : \tau}{\Sigma_{11}, \Sigma_{12} \rhd s \uplus \{\ell \mapsto v\} : \Sigma_2, \ell{:}\tau}$$

$\boxed{\rhd s; e : \tau}$ (configuration typing)

$$\textsc{Conf}\ \frac{\Sigma_1 \rhd s : \Sigma_1, \Sigma_2 \qquad \cdot; \Sigma_2 \rhd e : \tau}{\rhd s; e : \tau}$$

Figure 10. Statics (v): stores and configurations

Alms does just that. In §5 we show that our implicit promotion mechanism selects the best usage qualifier for function types.

Store and configuration judgments. In order to prove our type soundness theorem, we need to lift our typing judgments to stores and run-time configurations.

The type of a store is a typing context containing the names of the store's locations and the types of their contents. The store typing judgment $\Sigma_1 \rhd s : \Sigma_2$ gives store s type Σ_2 in the context of Σ_1, which is necessary because values in the store may refer to other values in the store. Rule S-CONS shows that the resources represented by context Σ_1 (*i.e.,* Σ_{11}, Σ_{12}) are split between the values in s.

Our preservation lemma concerns typing judgments on configurations, $\rhd s; e : \tau$, which means that e has type τ in the context of store s. To type the configuration by rule CONF, we type the store, splitting its type into Σ_1, which contains locations referenced from the store, and Σ_2, which contains locations referenced from e.

5. Theoretical Results

We now state our two main theorems—principal qualifiers and type soundness—and sketch their proofs. Full versions of our proofs may be found in an extended version of this paper available at www.ccs.neu.edu/~tov/pubs/alms.

Principal qualifiers. Alms and ${}^{a}\lambda_{ms}$ go to a lot of trouble to find the best usage qualifier expressions for function types. To make programming with affine types as convenient as possible, we want to maximize polymorphism between one-use and unlimited versions of functions. While writing the Alms standard library, we found that usage qualifier constants **A** and **U**, even with dereliction subtyping, were insufficient to give a principal type to some terms.

For example, consider function *default*, an eliminator for option types, sans function argument types:

```
let default (def: ...) (opt: ...) =
  match opt with
  | Some x → x
  | None   → def
```

Without usage qualifier expressions, *default* has at least two incomparable types:

$$\mathit{default}_1 : \forall\hat{\alpha}.\,\hat{\alpha} \xrightarrow{\mathsf{U}} \hat{\alpha}\ \mathsf{option} \xrightarrow{\mathsf{A}} \hat{\alpha}$$
$$\mathit{default}_2 : \forall\alpha.\,\alpha \xrightarrow{\mathsf{U}} \alpha\ \mathsf{option} \xrightarrow{\mathsf{U}} \alpha.$$

In the first case, because $\hat{\alpha}$ *might* be affine, the partial application of $\mathit{default}_1$ must be a one-use function, but in the second case we know that α is unlimited so partially applying $\mathit{default}_2$ and reusing the result is safe. Formally, these types are incomparable because the universally-quantified type variable $\hat{\alpha}$ in the former has a different kind than α in the latter, and Alms uses the *kernel* variant of rule TSUB-ALL. However, even were we to replace

rule TSUB-ALL with a rule analogous to $F^{\omega}_{<:}$'s *full* variant,

$$\frac{\Gamma, \alpha{:}\kappa \vdash \tau_1 <:^{\mathfrak{v}} \tau_2 \qquad \Gamma, \alpha{:}\kappa \vdash \kappa_1 <:^{-\mathfrak{v}} \kappa_2}{\Gamma \vdash \forall \alpha{:}\kappa_1.\, \tau_1 <:^{\mathfrak{v}} \forall \alpha{:}\kappa_2.\, \tau_2}\ \text{TSUB-ALL}_{\text{FULL}},$$

the types would not be related by the subtyping order. More importantly, neither type is preferable in an informal sense. The type of $\mathit{default}_1$ allows $\hat{\alpha}$ to be instantiated to an affine or unlimited type, but the result of partially applying it is a one-use function even if $\hat{\alpha}$ is known to be unlimited:

$$\begin{array}{ll} \mathit{default}_1\ 5 & : \mathsf{int\ option} \xrightarrow{\mathsf{A}} \mathsf{int} \\ \mathit{default}_1\ (\mathit{aref}\ 5) & : \mathsf{int\ aref\ option} \xrightarrow{\mathsf{A}} \mathsf{int\ aref}. \end{array}$$

If we choose $\mathit{default}_2$, the result of partial application is unlimited, but attempting to instantiate α to an affine type is a type error:

$$\begin{array}{ll} \mathit{default}_2\ 5 & : \mathsf{int\ option} \xrightarrow{\mathsf{U}} \mathsf{int} \\ \mathit{default}_2\ (\mathit{aref}\ 5) & : \textit{Type error!} \end{array}$$

Alms avoids both problems and instead discovers that the best usage qualifier for the arrow is the kind of the type variable:

$$\begin{array}{ll} \mathit{default} & : \forall \hat{\alpha}.\, \hat{\alpha} \xrightarrow{\mathsf{U}} \hat{\alpha}\ \mathsf{option} \xrightarrow{\langle \hat{\alpha} \rangle} \hat{\alpha} \\ \mathit{default}\ 5 & : \mathsf{int\ option} \xrightarrow{\mathsf{U}} \mathsf{int} \\ \mathit{default}\ (\mathit{aref}\ 5) & : \mathsf{int\ aref\ option} \xrightarrow{\mathsf{A}} \mathsf{int\ aref}. \end{array}$$

Because this is an important property, we prove a theorem that every typeable ${}^{a}\lambda_{ms}$ function has a principal usage qualifier.

Theorem 3 (Principal qualifiers)**.** *If* $\Gamma; \Sigma \rhd \lambda x{:}\tau.\, e : \tau_1 \xrightarrow{\varphi} \tau_2$*, then it has a least qualifier expression* φ_0*; that is,*

- $\Gamma; \Sigma \rhd \lambda x{:}\tau.\, e : \tau_1 \xrightarrow{\varphi_0} \tau_2$ *and*
- $\Gamma \vdash \varphi_0 <: \varphi'$ *for all* φ' *such that* $\Gamma; \Sigma \rhd \lambda x{:}\tau.\, e : \tau_1 \xrightarrow{\varphi'} \tau_2$.

Proof sketch. We obtain the principal qualifier φ_0 as follows. Let Σ_0 be the restriction of Σ to exactly the free variables and locations of $\lambda x{:}\tau.\, e$. Let φ_0 be the unique bound of Σ_0 given by $\Gamma \vdash \Sigma_0 \preceq \varphi_0$. By strengthening, $\Gamma; \Sigma_0 \rhd \lambda x{:}\tau.\, e : \tau_1 \xrightarrow{\varphi_0} \tau_2$, and by rule T-WEAK we can get the same type in Σ.

A derivation of $\Gamma; \Sigma \rhd \lambda x{:}\tau.\, e : \tau_1 \xrightarrow{\varphi'} \tau_2$ always involves rule T-ABS using some portion of Σ, followed by some number of subsumptions and weakenings. Subsumption will never let φ' be less than φ_0. However, weakening might allow us to type $\lambda x{:}\tau.\, e$ with a different portion of Σ than Σ_0. We know that any superset of Σ_0 has bound no less than φ_0, and while a non-superset of Σ_0 may have a smaller bound, we chose Σ_0 so that only Σ_0 and its supersets are suitable to type the term and then weaken to Σ. □

Thus, for a function in any given context, there is a least usage qualifier, and our implementation can find the least qualifier by considering only the portion of Σ that pertains to the free identifiers of the λ term, as suggested by the algorithmic rule

$$\text{(T-ABS}_{\text{ALG}}\text{)}\quad \frac{\vdash (\Gamma; \Sigma), x{:}\tau_1 \rightsquigarrow \Gamma'; \Sigma' \qquad \Gamma'; \Sigma' \rhd e : \tau_2 \qquad \varphi_0 = \begin{cases} \mathsf{A} & \text{if } \mathrm{locs}(e) \neq \varnothing \\ \bigsqcup \{\varphi \mid x \in \mathrm{fv}(e), \Gamma \vdash \Sigma(x) : \varphi\} & \text{otherwise} \end{cases}}{\Gamma; \Sigma \rhd \lambda x{:}\tau_1.\, e : \tau_1 \xrightarrow{\varphi_0} \tau_2}.$$

Type soundness. The key obstacle in our type soundness proof is establishing a substitution lemma, which in turn relies on showing that the kind of the type of any value accurately reflects the resources contained in that value, which itself comes as a corollary to the proposition that the kinds of subtypes are themselves subkinds:

Lemma 4 (Monotonicity of kinding)**.** *If* $\Gamma \vdash \tau_1 <:^{+} \tau_2$ *where* $\Gamma \vdash \tau_1 : \varphi_1$ *and* $\Gamma \vdash \tau_2 : \varphi_2$*, then* $\Gamma \vdash \varphi_1 <: \varphi_2$.

This lemma is the reason for the premise in rule OK-ARR that for a kind $\Pi\alpha^{\mathfrak{v}}.\, \kappa$, variance $\mathfrak{v}$ must be at least $+$ if $\alpha \in \mathrm{ftv}(\kappa)$. Otherwise, we could construct a counterexample to lemma 4:

- $\beta{:}\Pi\alpha^{-}.\, \langle\alpha\rangle \vdash \beta\ (1 \xrightarrow{\mathsf{A}} 1) <:^{+} (1 \xrightarrow{\mathsf{U}} 1)$,
- $\beta{:}\Pi\alpha^{-}.\, \langle\alpha\rangle \vdash \beta\ (1 \xrightarrow{\mathsf{A}} 1) : \mathsf{A}$, and
- $\beta{:}\Pi\alpha^{-}.\, \langle\alpha\rangle \vdash \beta\ (1 \xrightarrow{\mathsf{U}} 1) : \mathsf{U}$,
- but $\beta{:}\Pi\alpha^{-}.\, \langle\alpha\rangle \vdash \mathsf{A} <: \mathsf{U}$ is not the case.

The kind well-formedness judgment rules out kinds like $\Pi\alpha^{-}.\, \langle\alpha\rangle$.

Proof for lemma 4. We define an extension of the subkinding relation, $\Gamma \vdash \kappa_1 \lessapprox \kappa_2$, which is insensitive to the variances decorating Π kinds. Observe that on qualifier expressions this new relation coincides with subkinding. We generalize the induction hypothesis—if $\Gamma \vdash \tau_1 <:^{+} \tau_2$ where $\Gamma \vdash \tau_1 : \kappa_1$ and $\Gamma \vdash \tau_2 : \kappa_2$, then $\Gamma \vdash \kappa_1 \lessapprox \kappa_2$—and complete the proof by induction on the structure of the subtyping derivation. □

Corollary 5 (Kinding finds locations)**.** *Suppose that* $\Gamma; \Sigma \rhd v : \tau$ *and* $\Gamma \vdash \tau : \varphi$ *where* $\mathrm{dom}(\Sigma)$ *contains only locations* (ℓ)*. If any locations appear in* v *then* $\Gamma \vdash \mathsf{A} <: \varphi$.

Proof sketch. By induction on the typing derivation. We use the previous lemma in the case for the subsumption rule T-SUBSUME:

$$\textbf{Case}\ \frac{\Gamma; \Sigma \rhd v : \tau' \qquad \Gamma \vdash \tau' <:^{+} \tau \qquad \Gamma \vdash \tau : \varphi}{\Gamma; \Sigma \rhd v : \tau}.$$

By the induction hypothesis, $\Gamma \vdash \tau' : \mathsf{A}$, and by lemma 4, $\Gamma \vdash \mathsf{A} <: \varphi$. □

Corollary 5 lets us prove our substitution lemma. Then progress, preservation, and type soundness are standard:

Theorem 6 (Type soundness)**.** *If* $\rhd \{\}; e : \tau$ *then either* e *diverges or there exists some store* s *and value* v *such that* $\{\}; e \longmapsto^{*} s; v$ *and* $\rhd s; v : \tau$.

6. Related Work

In prior work, we showed how an Alms-like affine language may safely interoperate with a conventional (non-affine) language (Tov and Pucella 2010). In particular, the languages may freely share values, including functions. Attempts by the conventional language to subvert the affine language's invariants are prevented by dynamic checks in the form of behavioral software contracts. That paper focused specifically on multi-language interaction, using a predecessor of Alms.

System F°. Mazurak et al. (2010) describe a calculus of "lightweight linear types." Their primary motivation is similar to ours: to remove needless overhead and provide a "simple foundation for practical linear programming."

System F° and the prior iteration of Alms independently introduced several new ideas:

- Both use kinds to distinguish linear (in Alms, affine) types from unlimited types, where F°'s kinds $\circ$ and $\star$ correspond to our A and U, and their subkinding relation $\star \leq \circ$ corresponds to our $\mathsf{U} \sqsubseteq \mathsf{A}$.
- F° uses existentials and subkinding to abstract unlimited types into linear types. Alms (the language) uses modules and ${}^{a}\lambda_{ms}$ (the calculus) uses higher-kinded type abstraction to define abstract affine types, including type constructors with parameters. Mazurak et al. mention the possibility of extending F° with abstraction over higher kinds but do not show the details.

- They sketch out a convenient notation for writing linear computations. This inspired our different implicit threading syntax, which is implemented in Alms as mentioned at the end of §2.

There are also notable differences:

- F$^\circ$ has linear types, which disallow weakening, whereas Alms has affine types, which support it. This is a trade-off. Linear types make it possible to enforce liveness properties, which may be useful, for instance, to ensure that manual memory management does not leak. On the other hand, we anticipate that safely combining linearity with exceptions requires a type-and-effect system to track when raising an exception would implicitly discard linear values. Alms can support explicit deallocation so long as failure to do so is backed up by a garbage collector.
- Alms's unlimited-use function type is a subtype of its one-use function type. F$^\circ$ does not provide subtyping, though they do show how η expansion can explicitly perform the coercion that our subtyping does implicitly. Experience with our implementation confirms that dereliction subtyping is valuable, though we admit it comes at the cost of complexity.
- F$^\circ$ requires annotating abstractions ($\lambda^\kappa x{:}\tau.e$) to specify the kind of the resulting arrow type, which may only be $\star$ or $\circ$. Alms refines this with qualifier expressions and selects the least kind automatically.
- Mazurak et al. give a resource-aware semantics and prove that they can encode regular protocols. We do neither but conjecture that our system enjoys similar properties, except that weakening makes it possible to bail out of a protocol at any point.
- Their sketch of rules for algebraic datatypes is similar to how ours work, though ours are strictly stronger. For example, an option type in F$^\circ$ would have two versions:

  $$\mathsf{optionLin} : \circ \Rightarrow \circ \qquad \mathsf{optionUn} : \star \Rightarrow \star.$$

  Our dependent kinds in Alms let us define one type constructor whose kind subsumes both:

  $$\mathsf{option} : \Pi\hat{\alpha}^+.\, \langle\hat{\alpha}\rangle.$$

Clean. At first glance, Clean's uniqueness types *appear* to be dual to affine types. Uniqueness types are descriptive—they indicate that a particular reference is unique—while affine (and linear) types are prescriptive, since they restrict what may be done to some reference in the future but do not necessarily know where it's been. Similarly, Clean's subtyping relation, which allows forgetting that a value is unique, appears dual to Alms's, which allows pretending that an unlimited value is affine. However, the duality breaks down in the higher-order case. When a partially applied function captures some unique free variable, Clean's type system must *prohibit* aliasing of the function in order to maintain descriptive uniqueness when the function is fully applied (Plasmeijer and Eekelen 2002). In Clean's terminology, function types with the unique attribute are "essentially unique," but we might call them "affine."

There is a strong similarity between our kinding judgment and Clean's uniqueness propagation rules that relate the uniqueness of data structures to that of their constituent parts. While Clean supports subtyping, it does not have a subkinding relation analogous to Alms or F$^\circ$'s. In particular, Clean requires that the uniqueness attributes declared for an abstract type in a module's interface exactly match the uniqueness attributes in the module's implementation.

Use types and qualifiers. Wadler (1991) discusses several variants of linear type systems. He proposes something akin to dereliction subtyping (*i.e.,* $!A \leq A$) and points out that in such a system, terms such as $\lambda f.\lambda x.f\,x$ have several unrelated types. (We made a similar observation in §5.) In order to recover principal types, he introduces use types, which decorate the exponential modality with a *use variable* i: $!^i$. The use variable ranges over 0 and 1, where $!^0 A = A$ and $!^1 A = !A$. This provides principal types, but at the cost of adding use-variable inequality constraints to type schemes.

A use-variable inequality of the form $i \leq j$ is essentially an implication $i \supset j$, where 1 is truth and 0 is falsity. De Vries et al. (2008) show, in the setting of uniqueness types, how such inequalities may be represented instead using Boolean logic. For example, if we have a type

$$\ldots\ !^i\ \ldots\ !^j\quad \ldots, [i \leq j],$$

we can discard the inequality constraint and represent it instead as

$$\ldots\ !^i\ \ldots\ !^{i \vee k}\ \ldots,$$

because $i \leq (i \vee k)$. In general, any collection of use-variable inequalities (or uniqueness-attribute constraints) may be eliminated by replacing some of the use variables with propositional formulae over use variables. This insight is the source of Alms's usage qualifier expressions.

If we follow use types to their logical conclusion, we reach λ^{URAL} (Ahmed et al. 2005), wherein each type is composed of a pretype that describes its representation and a qualifier that gives its usage. Alms does not follow this approach because we insist that qualified types are too verbose for a user-visible type system. Their system's qualifier lattice includes two more than ours, R for relevant types that allow duplication but not discarding, and L for linear types. This results in a rich and elegant system, but we do not believe R and L would be useful in a language like Alms.

However, there is an interesting correspondence between our kinding rules and their type rules. For example, our product type constructor ($\times$) has kind $\Pi\alpha^+.\Pi\beta^+.\langle\alpha\rangle \sqcup \langle\beta\rangle$, which means that the kind of a product type is the least upper bound of the kinds of its components. The product typing rule in λ^{URAL} enforces a similar constraint, that the qualifier of a product type, ξ, must upper bound the qualifiers of its components τ_1 and τ_2.

$$\frac{\begin{array}{cc}\Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 & \Delta \vdash \xi : \mathsf{QUAL} \\ \Delta; \Gamma_1 \vdash v_1 : \tau_1 & \Delta \vdash \tau_1 \preceq \xi \\ \Delta; \Gamma_2 \vdash v_2 : \tau_2 & \Delta \vdash \tau_2 \preceq \xi\end{array}}{\Delta; \Gamma \vdash \langle v_1, v_2\rangle : {}^{\xi}(\tau_1 \otimes \tau_2)}\ (\textsc{MPair}).$$

Vault. DeLine and Fähndrich's Vault (2001) is a safe, low-level language with support for typestate. It tracks *keys*, which associate static capabilities with the identity of run-time objects, in the same manner that Alms uses existentially-quantified type variables to tie values to capabilities. This allows static enforcement of a variety of protocols. As an example, DeLine and Fähndrich give a tracked version of the Berkeley Sockets API. In previous work on Alms we show how Alms expresses the same interface.

Vault's treatment of capabilities may be more convenient to use than Alms's, because while Alms requires explicit threading of capability values, Vault's key sets are tracked automatically within function bodies. On the other hand, because capabilities in Alms appear as ordinary values, we may combine them using the native intuitionistic logic of Alms's type system. Instead, Vault must provide a simple predicate calculus for expressing pre- and post-conditions. For more complicated logic, Vault allows embedding capabilities in values, but since the values are untracked, extracting a capability from a value requires a dynamic check. Alms's type system eliminates the need for such checks for affine values stored in algebraic datatypes, though it also allows dynamic management of affine values by storing them in reference cells.

Notably, Alms can also express Fähndrich and DeLine's *adoption and focus* (2002).

Sing#. Microsoft's experimental Singularity operating system is written in Sing#, a high-level systems programming language that extends Spec# (Fähndrich et al. 2006). Sing# has built-in support for *channel contracts,* which are a form of session type providing static checking of communication protocols between device drivers and other services. Unlike more idealistic linear systems, the design acknowledges the need to allow for failure: every protocol implicitly includes branches to close the channel at any point.

Sing# processes do not share memory but can allocate tracked objects on a common *exchange heap*. Only one process has access to an exchange heap object at a given time, but a process may give up access and transmit the object over a channel to another process, which then claims ownership of it.

Alms's library includes two different implementations of session types supporting different interfaces, and the exchange heap concept is easily expressible as well.

7. Future Work and Conclusion

We already enjoy programming in Alms, but we are not done yet.

Unit subsumption. In §2, we found that adding capabilities to an existing interface often involves wrapping the old version of a function to ignore a new argument of type unit or construct a tuple containing unit for its result. This is unnecessary. While the client outside the abstraction barrier needs to see types that involve the affine capabilities, the implementation has no use for them.

To eliminate much of this noise, we can extend our subtyping relation to take advantage of the fact that unit is, well, a unit:

$$\frac{\Gamma \vdash \tau_2 \text{ singleton}}{\Gamma \vdash \tau_1 <:^{\mathfrak{v}} \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash \tau_2 \text{ singleton}}{\Gamma \vdash \tau_1 <:^{\mathfrak{v}} \tau_2 \times \tau_1}$$

$$\frac{\Gamma \vdash \tau_2 \text{ singleton}}{\Gamma \vdash \tau_1 \xrightarrow{\varphi} \tau_2 \xrightarrow{\varphi} \tau_3 <:^{\mathfrak{v}} \tau_1 \xrightarrow{\varphi} \tau_3}$$

This is implementable via a type erasure technique such as intensional polymorphism (Crary et al. 2002). Not representing compile-time capabilities at run time has performance benefits as well.

Type inference. Alms's local type inference eliminates most explicit type applications, but needing to annotate all function arguments is irksome. To fix this, we are exploring possibilities for type inference. While we suspect that our limited subtyping should not impede full Damas-Milner–style inference (Damas and Milner 1982), Alms has several idioms that rely on existential types. We are exploring whether an extension for first-class polymorphism, such as HML (Leijen 2009), would be suitable for Alms.

Alms is not finished, but our prototype is at this point usable for experimentation. It is based on a calculus, ${}^{a}\lambda_{ms}$, whose type system we have proved sound. While some parts of the type system are complex, we have seen in practice that Alms types are tractable and Alms programs do not look very different from the functional programs to which we are accustomed. It currently implements algebraic datatypes, exceptions, pattern matching, concurrency, and opaque signature ascription. The language is rich enough to express Vault-style typestate, a variety of static and dynamic locking protocols, checked downcasts of one-use functions to unlimited-use functions, session types, and more.

Acknowledgments

We wish to thank J. Daniel Brown, Bryan Chadwick, Matthias Felleisen, Sam Tobin-Hochstadt, Vincent St-Amour, and the anonymous referees for their helpful comments, discussion, and corrections. This research was supported in part by AFOSR grant FA9550-09-1-0110.

References

A. Ahmed, M. Fluet, and G. Morrisett. A step-indexed model of substructural state. In *ICFP'05*, pages 78–91. ACM, 2005.

A. Barber. Dual intuitionistic linear logic. Technical Report ECS-LFCS-960347, U. of Edinburgh, 1996.

P. S. Barth, R. S. Nikhil, and Arvind. M-structures: Extending a parallel, non-strict, functional language with state. In *Proc. FPCA'91*, volume 523 of *LNCS*. Springer, 1991.

G. M. Bierman. *On Intuitionistic Linear Logic*. PhD thesis, U. of Cambridge, 1993.

J. Boyland. Checking interference with fractional permissions. In *SAS'03*, volume 2694 of *LNCS*. Springer, 2003.

K. Crary, S. Weirich, and G. Morrisett. Intensional polymorphism in type-erasure semantics. *JFP*, 12(06):567–600, 2002.

L. Damas and R. Milner. Principal type-schemes for functional programs. In *POPL'82*, pages 207–212. ACM, 1982.

R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI'01*. ACM, 2001.

M. Fähndrich and R. DeLine. Adoption and focus: Practical linear types for imperative programming. In *PLDI'02*. ACM, 2002.

M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in Singularity OS. In *EuroSys'06*, pages 177–190. ACM, 2006.

S. J. Gay, V. T. Vasconcelos, and A. Ravara. Session types for inter-process communication. Technical Report TR-2003-133, U. of Glasgow, 2003.

J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

D. Leijen. Flexible types: Robust type inference for first-class polymorphism. In *POPL'09*, pages 66–77. ACM, 2009.

X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. *The Objective Caml system*. INRIA, 3.11 edition, 2008.

K. Mazurak, J. Zhao, and S. Zdancewic. Lightweight linear types in System F°. In *TLDI'10*, pages 77–88. ACM, 2010.

R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML*. MIT, revised edition, 1997.

B. C. Pierce. *Types and Programming Languages*. MIT, 2002.

R. Plasmeijer and M. van Eekelen. *Clean Language Report, Version 2.1*. Dept. of Software Technology, U. of Nijmegen, 2002.

A. Rossberg, C. V. Russo, and D. Dreyer. F-ing modules. In *TLDI'10*, pages 89–102. ACM, 2010.

M. Steffen. *Polarized Higher Order Subtyping*. PhD thesis, Universität Erlangen-Nürnberg, 1997.

I. E. Sutherland and G. W. Hodgman. Reentrant polygon clipping. *CACM*, 17(1):32–42, 1974.

J. A. Tov and R. Pucella. Stateful contracts for affine types. In *ESOP'10*, volume 6012 of *LNCS*, pages 550–569. Springer, 2010.

E. de Vries, R. Plasmeijer, and D. M. Abrahamson. Uniqueness typing simplified. In *IFL'07*, pages 201–218. Springer, 2008.

P. Wadler. Is there a use for linear logic? In *PEPM'91*, pages 255–273. ACM, 1991.

P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL'89*, pages 60–76. ACM, 1989.