

Short Presentation: Linear Types for Aliased Resources

Chris Hawblitzel
Microsoft Research

1. INTRODUCTION

Traditional type systems allow unrestricted duplication of values. Given a variable x of type *file*, a traditionally typed program may instantiate a variable y with a copy of x :

```
let f(x : file) = (let y = x in close(x); write(y, "hello"))
```

In this example, x and y are *aliases* — they both refer to the same file. Unfortunately, this aliasing causes a traditional type checker to miss the mistake in the code above, where the *write* attempts to write to a closed file. By contrast, a *linear* type system would catch the mistake in the code above, because a linear variable x would be consumed by the instantiation $y = x$, and would therefore be out of scope in the expression *close(x)*. Nevertheless, linear types are often considered too restrictive to use in practice, because many programs rely on aliasing, and linear types appear to prohibit aliasing entirely. Because of this apparent prohibition, Crary, Walker, and Morrisett [1] proposed a more complicated type system, called the calculus of capabilities, that augments simple linear types with *duplicable* types. We argue that linear types can express aliasing, and that in fact, a very simple, standard, decidable linear logic can express all of the capability calculus's rules for linear and duplicable types. The complete encoding of the capability calculus using linear types is found in [2]; here, we present some examples of how linear types express aliasing, along with a brief overview of the complete encoding.

The capability calculus describes the state of linear resources using a *capability*, drawn from the following syntax:

$$C = \epsilon \mid \emptyset \mid \{\alpha^1\} \mid \{\alpha^+\} \mid C_1 \oplus C_2 \mid \overline{C}$$

Each resource α in a capability may be specified as unique ($\{\alpha^1\}$), meaning that it appears nowhere else in the capability, or duplicable ($\{\alpha^+\}$), meaning that it may appear elsewhere in the capability. The join operator \oplus combines capabilities together, so that a single capability may describe many resources. Joins are idempotent for duplicable resources ($\{\alpha^+\} = \{\alpha^+\} \oplus \{\alpha^+\}$) but not for unique resources ($\{\alpha^1\} \neq \{\alpha^1\} \oplus \{\alpha^1\}$).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SPACE 2006 Charleston, South Carolina USA
Copyright ACM ...\$5.00.

$C \vdash C$	$\vdash \emptyset$	$\frac{\Lambda \vdash C}{\Lambda, \emptyset \vdash C}$	$\Lambda \vdash \text{true}$
$\frac{\Lambda_1 \vdash C_1 \quad \Lambda_2 \vdash C_2}{\Lambda_1, \Lambda_2 \vdash C_1 \otimes C_2}$		$\frac{\Lambda, C_1, C_2 \vdash C_3}{\Lambda, C_1 \otimes C_2 \vdash C_3}$	
$\frac{\Lambda \vdash C_1 \quad \Lambda \vdash C_2}{\Lambda \vdash C_1 \& C_2}$		$\frac{\Lambda, C_k \vdash C_3}{\Lambda, C_1 \& C_2 \vdash C_3} (k \in \{1, 2\})$	
$\frac{\Lambda, C_1 \vdash C_2}{\Lambda \vdash C_1 \multimap C_2}$		$\frac{\Lambda_1 \vdash C_1 \quad \Lambda_2, C_2 \vdash C_3}{\Lambda_1, \Lambda_2, C_1 \multimap C_2 \vdash C_3}$	

Figure 1: Linear sequent rules (where $\Lambda = C_1, \dots, C_n$)

Our encoding translates capabilities from the syntax above into linear logic formulas drawn from the following syntax:

$$C = \epsilon \mid \emptyset \mid \{\alpha\} \mid C_1 \otimes C_2 \mid C_1 \& C_2 \mid C_1 \multimap C_2 \mid \text{true}$$

The logic contains formulas $C_1 \otimes C_2$ (“both C_1 and C_2 ”), $C_1 \& C_2$ (“choice of either C_1 or C_2 ”), and $C_1 \multimap C_2$ (“consuming C_1 produces C_2 ”), as well as variables ϵ , resources $\{\alpha\}$, and constants \emptyset (empty) and *true*. Figure 1 shows the inference rules for these linear logic formulas. Note that assumptions in linear logic cannot be duplicated or discarded; the assumption C, C is equivalent to neither C nor C, C, C (for an introduction to linear logic, see [6]).

Figure 2 shows examples of functions whose linear resource usage is specified by capability calculus capabilities (left side) and linear logic formulas (right side). (For simplicity, the examples show preconditions and postconditions, rather than using the low-level continuation-passing style types described in [1] and [2].) The first function, g , has the simplest specification: for any resource α , g requires α to be unique on entry to g , and g ensures that α is still unique upon exit. The next function, $h1$, expects two linear resources β and γ , but closes β , so that the postcondition only mentions γ . Because both $\{\beta^1\}$ and $\{\gamma^1\}$ are unique, they cannot refer to the same file (β and γ cannot be aliased). This means that the implementation of g would be unable to call $h1$ with both $\beta = \alpha$ and $\gamma = \alpha$ — this would create a precondition $\{\alpha^1\} \oplus \{\alpha^1\}$, which is not equivalent to $\{\alpha^1\}$. Indeed, this restriction saves g from making a serious mistake; if $h1$ closes the resource β (which g instantiated with α), then it would be incorrect for g 's postcondition to claim that α is still alive.

$g[\alpha] : pre\{\alpha^1\}, post\{\alpha^1\}$ $h1[\beta, \gamma] : pre\{\beta^1\} \oplus \{\gamma^1\}, post\{\gamma^1\}$ $h2[\beta, \gamma] : pre\{\beta^+\} \oplus \{\gamma^+\}, post\{\beta^+\} \oplus \{\gamma^+\}$ $h3[\beta, \gamma, (\epsilon \leq \{\beta^+\} \oplus \{\gamma^+\})] : pre \epsilon, post \epsilon$ $h4[\beta, \gamma, \epsilon] : pre \epsilon \oplus \{\beta^1\} \oplus \{\gamma^+\}, post \epsilon \oplus \{\gamma^+\}$ $h5[\beta, \gamma, \delta, \epsilon \leq \{\delta^+\}] : pre \epsilon \oplus \{\beta^1\} \oplus \{\gamma^+\},$ $post \epsilon \oplus \{\gamma^+\}$	$g[\alpha] : pre\{\alpha\}, post\{\alpha\}$ $h1[\beta, \gamma] : pre\{\beta\} \otimes \{\gamma\}, post\{\gamma\}$ $h2[\beta, \gamma] : pre(\{\beta\} \otimes true) \& (\{\gamma\} \otimes true),$ $post(\{\beta\} \otimes true) \& (\{\gamma\} \otimes true)$ $h3[\beta, \gamma, \epsilon] : pre \epsilon \& (\{\beta\} \otimes true) \& (\{\gamma\} \otimes true),$ $post \epsilon \& (\{\beta\} \otimes true) \& (\{\gamma\} \otimes true)$ $h4[\beta, \gamma, \epsilon', \epsilon''] : pre \epsilon' \otimes \{\beta\} \otimes (\epsilon'' \& (\{\gamma\} \otimes true)),$ $post \epsilon' \otimes (\epsilon'' \& (\{\gamma\} \otimes true))$ $h5[\beta, \gamma, \delta, \epsilon', \epsilon''] :$ $pre \epsilon' \otimes \{\beta\} \otimes (\epsilon'' \& (\epsilon' \multimap \{\delta\} \otimes true) \& (\{\gamma\} \otimes true))$ $post \epsilon' \otimes (\epsilon'' \& (\epsilon' \multimap \{\delta\} \otimes true) \& (\{\gamma\} \otimes true))$
---	--

Figure 2: Examples in capability calculus (left), linear logic (right)

By contrast, $h2$ does not require β and γ to be distinct, because the “+” annotation indicates that $\{\beta^+\}$ and $\{\gamma^+\}$ are duplicable. This means that g could call $h2$ with both $\beta = \alpha$ and $\gamma = \alpha$, since the capability calculus considers g ’s precondition $\{\alpha^1\}$ to be a subcapability of $\{\alpha^+\}$, which is in turn equivalent to $\{\alpha^+\} \oplus \{\alpha^+\}$. For safety’s sake, $h2$ is not allowed to deallocate β and γ , so $h2$ ’s postcondition will be the same as its precondition: $\{\alpha^+\} \oplus \{\alpha^+\}$ in the case where $\beta = \alpha$ and $\gamma = \alpha$. From g ’s perspective, this postcondition is too weak, though; g cannot coerce $\{\alpha^+\} \oplus \{\alpha^+\}$ back to $\{\alpha^1\}$. The function $h3$ solves this problem using *bounded quantification*. Rather than directly declaring $\{\beta^+\} \oplus \{\gamma^+\}$ to be its precondition, $h3$ allows any ϵ as its precondition, subject to the constraint that ϵ is a subcapability of $\{\beta^+\} \oplus \{\gamma^+\}$. For example, g may call $h3$ with $\beta = \alpha$ and $\gamma = \alpha$ and $\epsilon = \{\alpha^1\}$, which satisfies $\epsilon \leq \{\beta^+\} \oplus \{\gamma^+\}$, and gives g the strong postcondition $\{\alpha^1\}$ that it needs. This ability to fully recover the uniqueness of a resource α after temporarily aliasing α distinguishes the capability calculus from other approaches to linearity and aliasing¹, and allows the capability calculus to support memory management based on aliasable regions[1][5].

The linear logic versions of g and $h1$ are straightforward: unique resources $\{\alpha^1\}$ become linear logic resources $\{\alpha\}$, and joins of unique resources $\{\beta^1\} \oplus \{\gamma^1\}$ become linear pairs $\{\beta\} \otimes \{\gamma\}$. Linear logic’s equivalent of duplicable resources is less obvious, so consider a simple example: suppose the resource in question is a hand, where it takes one hand to wave and two hands to clap. Given $\{\beta^+\} \oplus \{\gamma^+\}$, it’s possible to wave β and wave γ as often as desired, but it’s never possible to clap β and γ together, because although $\{\beta^+\} \oplus \{\gamma^+\}$ gives access to either β or γ at any given time, it never gives access to two distinct resources β and γ simultaneously (since β and γ might be two aliases for the same resource, not two distinct resources). This suggests using linear logic’s choice operator $C_1 \& C_2$, which gives access to either C_1 or C_2 , but not both. The following judgments demonstrate this idea:

$$\frac{\begin{array}{l} \{\alpha\} \vdash (\{\alpha\} \otimes true) \& (\{\alpha\} \otimes true) \\ \{\beta\} \otimes \{\gamma\} \vdash (\{\beta\} \otimes true) \& (\{\gamma\} \otimes true) \end{array}}{\quad}$$

¹For example, extended L^3 [3] can “freeze” a unique resource to make it aliasable (even more aliasable, in fact, than the capability calculus’s duplicable capabilities), and then “thaw” and “refreeze” the resource, but can never get a frozen resource completely back to its original unique state — a resource can change its state τ before its first freeze, but after getting frozen at a particular τ , each subsequent refreeze requires the same τ .

The first judgment shows how g may call $h2$: by instantiating both β and γ with α , g uses its precondition $\{\alpha\}$ to satisfy $h2$ ’s precondition $(\{\beta\} \otimes true) \& (\{\gamma\} \otimes true)$. The second judgment shows how $h1$ may call $h2$. Notice the importance of the $\otimes true$ in the second judgment; the judgment $\{\beta\} \otimes \{\gamma\} \vdash \{\beta\} \& \{\gamma\}$ is not derivable in linear logic.

Subcapabilities present another challenge to the linear logic encoding. Here, the key insight [7][4] is that a subcapability relation $\epsilon \leq C$ gives anyone possessing ϵ a choice: either use take ϵ as it is, or coerce it to C (note that the capability calculus’s rules prohibit ϵ from appearing inside C in the bounded declaration $\epsilon \leq C$). Thus, the declaration $\epsilon \leq C$ may be replaced by a simple declaration of ϵ , and a replacement of all free occurrences of ϵ with $\epsilon \& C$, as shown in the linear logic version of $h3$. If g calls $h3$ with $\beta = \alpha$ and $\gamma = \alpha$ and $\epsilon = \{\alpha\}$, then g ’s precondition implies $h3$ ’s precondition and $h3$ ’s postcondition implies g ’s postcondition:

$$\begin{array}{l} \{\alpha\} \vdash \{\alpha\} \& (\{\alpha\} \otimes true) \& (\{\alpha\} \otimes true) \\ \{\alpha\} \& (\{\alpha\} \otimes true) \& (\{\alpha\} \otimes true) \vdash \{\alpha\} \end{array}$$

Similarly, $h1$ may call $h3$ by choosing $\epsilon = \{\beta\} \otimes \{\gamma\}$, which implies $h3$ ’s precondition and is implied by $h3$ ’s postcondition:

$$\begin{array}{l} \{\beta\} \otimes \{\gamma\} \vdash (\{\beta\} \otimes \{\gamma\}) \& (\{\beta\} \otimes true) \& (\{\gamma\} \otimes true) \\ (\{\beta\} \otimes \{\gamma\}) \& (\{\beta\} \otimes true) \& (\{\gamma\} \otimes true) \vdash \{\beta\} \otimes \{\gamma\} \end{array}$$

In these examples, both g and $h1$ call $h3$ by instantiating ϵ with purely unique capabilities ($\epsilon = \{\alpha^1\}$ for g and $\epsilon = \{\beta^1\} \oplus \{\gamma^1\}$ for $h1$). In general, though, capability variables may stand for mixtures of unique and duplicable capabilities. To see how such mixtures complicate the linear logic encoding, consider two potential encodings of $h4$ ’s precondition $\epsilon \oplus \{\beta^1\} \oplus \{\gamma^+\}$ and postcondition $\epsilon \oplus \{\gamma^+\}$:

$$\begin{array}{ll} pre (\epsilon \otimes \{\beta\}) \otimes (\{\gamma\} \otimes true), & post \epsilon \otimes (\{\gamma\} \otimes true) \\ pre \{\beta\} \otimes (\epsilon \& (\{\gamma\} \otimes true)), & post \epsilon \& (\{\gamma\} \otimes true) \end{array}$$

The first encoding supports unique ϵ well, but fails on a duplicable $\epsilon = \{\gamma^+\}$; the postcondition $(\{\gamma\} \otimes true) \otimes (\{\gamma\} \otimes true)$ is never satisfiable, for example. The second encoding supports aliased ϵ well, but fails on a unique $\epsilon = \{\alpha^1\}$; the postcondition $\{\alpha\} \& (\{\gamma\} \otimes true)$ is too weak to imply both $\{\alpha\}$ and $\{\gamma\}$ together. Neither encoding is sufficient in general, since $h4$ must be polymorphic over all instantiations of ϵ , including instantiations that mix unique and duplicable capabilities, such as $\epsilon = \{\alpha^1\} \oplus \{\gamma^+\}$. Therefore, the general encoding splits capability variables ϵ into separate unique and duplicable portions ϵ' and ϵ'' (where $\epsilon = \epsilon' \oplus \epsilon''$).

As shown in figure 2's encoding of function $h4$, the unique portion ϵ' resides with other unique resources (such as $\{\beta^1\}$ in $h4$), while the duplicable portion ϵ'' resides with the duplicable resources (such as $\{\gamma^+\}$ in $h4$). More generally, the encoding of a capability C consists of a unique encoding and a duplicable encoding paired together to form $\mathcal{U}(C) \otimes \mathcal{D}(C)$, where:

$$\begin{array}{ll} \mathcal{U}(\epsilon) = \epsilon' & \mathcal{D}(\epsilon) = \epsilon'' \\ \mathcal{U}(\emptyset) = \emptyset & \mathcal{D}(\emptyset) = \emptyset \\ \mathcal{U}(\{\alpha^1\}) = \{\alpha\} & \mathcal{D}(\{\alpha^1\}) = \emptyset \\ \mathcal{U}(\{\alpha^+\}) = \emptyset & \mathcal{D}(\{\alpha^+\}) = \{\alpha\} \\ \mathcal{U}(C_1 \oplus C_2) = & \mathcal{D}(C_1 \oplus C_2) = \\ & (\mathcal{D}(C_1) \otimes \text{true}) \\ & \&(\mathcal{D}(C_2) \otimes \text{true}) \end{array}$$

Splitting variables into two parts introduces a slight complication into the encoding of bounded quantification $\epsilon \leq C$, since $\epsilon' \oplus \epsilon'' \leq C$ no longer defines a single variable ϵ to replace with $\epsilon \& C$. To resolve this issue, observe that having ϵ'' gives you a choice: take ϵ'' as it is, or combine it with ϵ' to produce C . For a C that is purely duplicable², this suggests replacing free occurrences of ϵ'' with $\epsilon'' \& (\epsilon' \multimap C)$. The encoding of $h5$, for example, is the same as the encoding of $h4$, except that the constraint $\epsilon \leq \{\delta^+\}$ turns into $\epsilon' \oplus \epsilon'' \leq \{\delta^+\}$, which the encoding represents by substituting $\epsilon'' \& (\epsilon' \multimap \{\delta\} \otimes \text{true})$ for ϵ'' . If the caller chooses $\epsilon = \{\delta^1\}$ in the capability calculus, for instance, then in the linear logic encoding, the caller can choose $\epsilon' = \{\delta\}$ and $\epsilon'' = \emptyset$, so that $\epsilon'' \& (\epsilon' \multimap \{\delta\} \otimes \text{true})$ is $\emptyset \& (\{\delta\} \multimap \{\delta\} \otimes \text{true})$, which is trivially satisfied.

Finally, the capability calculus supports *stripped* capabilities \overline{C} , along with equivalence rules that say $\{\alpha^1\} = \{\alpha^+\}$, $\{\alpha^+\} = \{\alpha^+\}$, $\overline{C_1 \oplus C_2} = \overline{C_1} \oplus \overline{C_2}$, $\overline{\overline{C}} = \overline{C}$, and $\overline{\emptyset} = \emptyset$. In fact, the only capability without a simplifying equivalence rule is a variable ϵ ; it's not immediately obvious how to represent $\overline{\epsilon}$ without using the stripping operator. Therefore, for each variable ϵ , the encoding introduces a auxillary variable ϵ_S , and each time ϵ is instantiated with a capability C , the encoding instantiates ϵ_S with the encoding of \overline{C} . To preserve the capability calculus's equivalence rules (in particular, the rule $C \leq \overline{C}$), the encoding constrains each ϵ so that $\epsilon \leq \epsilon_S$.

Based on these ideas, the complete encoding [2] translates any well-typed capability calculus program into a well-typed program based on linear logic. The encoding first eliminates stripped capabilities \overline{C} , then splits variables into unique and duplicable portions and translates each $\epsilon \leq C$ using the $\epsilon'' \& (\epsilon' \multimap C)$ approach described above.

So far, this short paper has discussed the capability calculus's capability language but not the capability calculus's term language. In fact, although [2] explores some alternative source term languages and target term languages, it is not necessary to change the term language syntax at all to accommodate capabilities based on linear logic: the only changes required to the syntax are the new definition of C presented here and a simplified definition of type contexts Δ . This means that the term encoding (in contrast to the elaborate capability encoding) is nearly trivial: just introduce and instantiate extra variables ϵ_S , ϵ' , and ϵ'' for each

² A non-duplicable C requires more effort; [2] encodes $\epsilon \leq C$ by splitting C into its unique and duplicable portions C' and C'' , declaring another variable ϵ''' subject to the constraints $\epsilon' = \epsilon''' \oplus C'$ and $\epsilon'' \oplus \epsilon''' \leq C''$, and then substituting $\epsilon''' \oplus C'$ for ϵ' and $\epsilon'' \& (\epsilon''' \multimap C'')$ for ϵ'' .

ϵ introduction and instantiation. Furthermore, this ensures that the encoding introduces no run-time overhead, compared to the original capability calculus program. Finally, this means that it is easy to update the capability calculus's soundness proof to reflect the new definitions of C and Δ . The end result is a simpler language (compare the rules in figure 1 to the larger set of equivalence and subcapability rules found in the capability calculus) that is still sound, yet expresses any capability calculus program.

Acknowledgments: Thanks to David Walker, who suggested many of the ideas presented here [7].

2. REFERENCES

- [1] Karl Cravy, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Symposium on Principles of programming languages*, pages 262–275. ACM Press, 1999.
- [2] Chris Hawblitzel. Linear types for aliased resources (extended version). Technical Report MSR-TR-2005-141, Microsoft Research, 2005.
- [3] Greg Morrisett, Amal J. Ahmed, and Matthew Fluet. L^3 : A linear language with locations. In *TLCA*, pages 293–307, 2005.
- [4] Benjamin C. Pierce. *Programming with Intersection Types and Bounded Polymorphism*. PhD thesis, 1991.
- [5] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.
- [6] P. L. Wadler. A taste of linear logic. In *Proceedings of the 18th International Symposium on Mathematical Foundations of Computer Science, Gdansk, New York, NY, 1993*. Springer-Verlag.
- [7] David Walker. Mechanical reasoning about low-level programs. lecture notes, <http://www.cs.cmu.edu/~dpw/papers.html>, 2001.