Linear Types for Aliased Resources (Extended Version)

Chris Hawblitzel

October 2005

Technical Report
MSR-TR-2005-141

Type systems that track aliasing can verify state-dependent program properties. For example, such systems can verify that a program does not access a resource after deallocating the resource. The simplest way to track aliasing is to use linear types, which on the surface appear to ban the aliasing of linear resources entirely. Since banning aliasing is considered too draconian for many practical programs, researchers have proposed type systems that allow limited forms of aliasing, without losing total control over state-dependent properties. This paper describes how to encode one such system, the capability calculus, using a type system based on plain linear types with no special support for aliasing. Given well-typed capability calculus source programs, the encodings produce well-typed target programs based on linear types. These encodings demonstrate that, contrary to common expectations, linear type systems can express aliasing of linear resources.

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
http://www.research.microsoft.com

1 Introduction

Type systems verify many important properties of programs. A type-safe program cannot accidentally use a floating point number as a file, for instance, because files and floating-point numbers have different types. Some properties exceed the grasp of conventional type systems, though, because the properties depend on the state of a resource. Ideally, a program should not be able to write to a file after closing the file, but most type systems, including those of Java, C#, and ML, assign the same type to an open file and a closed file, and thus cannot distinguish between writing to an open file and writing to a closed file. As another example, typical safe languages simply ban explicit heap object deallocation rather than attempting to statically verify that a program never accesses a deallocated heap object.

Several recent languages [7][24][9][10][32] have extended conventional type systems to track the state of resources, such as files, sockets, hardware devices, regions, and heap objects. In these languages, the type of an object changes as the object's state changes. The central challenge in these languages is aliasing — if both x and y refer to the same resource, then both x's type and y's type must change if the resource's state changes.

As an example, consider a simple interface to a file system, together with two functions *hello* and *goodbye*, expressed in an ML-like notation:

$$\begin{array}{l} \mathit{open} : \mathit{string} \to \mathit{file} \\ \mathit{close} : \mathit{file} \to () \\ \mathit{write} : \mathit{file} \times \mathit{string} \to () \\ \mathit{hello} : \mathit{file} \times \mathit{file} \to () \\ \mathit{hello}\ (x_1 : \mathit{file}, x_2 : \mathit{file}) = \mathit{write}\ (x_1, \text{``}\mathit{hello}\text{''});\ \mathit{write}\ (x_2, \text{``}\mathit{hello}\text{''}) \\ \mathit{goodbye} : \mathit{file} \times \mathit{file} \to () \\ \mathit{goodbye}\ (x_1 : \mathit{file}, x_2 : \mathit{file}) = \mathit{write}\ (x_1, \text{``}\mathit{bye}\text{''});\ \mathit{close}\ x_1;\ \mathit{write}\ (x_2, \text{``}\mathit{bye}\text{''});\ \mathit{close}\ x_2 \end{array}$$

The function *goodbye* behaves incorrectly if x_1 and x_2 refer to the same file, because it tries to write to x_2 after closing x_1:

$$\begin{array}{l} \mathit{byebye} : \mathit{file} \to () \\ \mathit{byebye}\ (x : \mathit{file}) = \mathit{goodbye}\ (x, x) \end{array}$$

In general, a type system's ability to track object state is limited by its ability to track aliasing: unless the type system knows that *goodbye* requires x_1 and x_2 to refer to distinct files, it cannot know that "*goodbye* (x, x)" is a mistake.

Linear type systems [27][28] deal with aliasing in a simple way: programs are not allowed to duplicate nor discard linear values (values having linear type). If file is a linear type, for example, then the function *byebye* is ill-typed because it tries to duplicate the linear variable x.

At first glance, it might seem that linear type systems prohibit the aliasing of linear values entirely, which is a rather draconian way to track aliasing. In particular, it is now impossible to call *hello* with arguments x_1 and x_2 bound to the same file, even though it would be safe to do so. This doesn't mean that it's impossible to write traditional nonlinear functions at all in a linear type system; all practical linear type systems provide both linear and nonlinear types (one common notation uses "!" to indicate nonlinearity; if file is a linear type, then $!\mathit{file}$ is the corresponding nonlinear type), so *hello* could declare its arguments to be nonlinear files. What seems to be missing from linear type systems is a middle ground between linearity and nonlinearity; the aliasing of linear values is prohibited entirely, and the aliasing of nonlinear values is completely unrestricted.

This problem motivated the development of the capability calculus [7] and alias types [24], which allow linear values to become aliased temporarily, but can later recover the linearity of the aliased values. For example, a function in these systems can pass a single linear file as both the x_1 and x_2 arguments to *hello*, and later recover the linearity of the file in order to close the file. Unfortunately, these systems require some fairly esoteric typing

constructs, including non-idempotent capability joins, subcapabilities, bounded capability quantification, and a special stripping operator. This paper demonstrates that these esoteric features are not necessary to achieve the power of the capability calculus and alias types. In fact, standard linear type systems, with no special extensions, can encode the capability calculus and alias types.

To substantiate this claim, this paper takes the capability calculus of Crary, Walker, and Morrisett (modified to handle generic linear "resources", rather than memory regions) as a source language, compiles it to two linear target languages, and proves that the compilation translates any well-typed source program into well-typed target programs. The first target language is identical to the source language except for the type system, which replaces the unusual features of the capability calculus with a straightforward, decidable linear logic. Since the compilation to the first target language affects only the types, it introduces no run-time overhead. The second target language is the standard polymorphic lambda calculus $F\omega$, extended with linear types. While the compilation to linear $F\omega$ does not produce efficient programs, it does demonstrate that the non-trivial aliasing present in the capability calculus and alias types is expressible even in a minimal linear type system. Furthermore, the compilation to linear $F\omega$ provides a minimal semantic basis for alias types. Finally, this paper also translates the unmodified capability calculus (with memory regions) into a third target language, which supports memory regions but uses the first target language's decidable logic.

By replacing the capability calculus's types with standard linear types, these three compilations reduce the complexity of the type system and thus reduce the trusted computing base of systems that rely on type checking for security, such as typed assembly language[19]. Furthermore, the techniques in this paper increase the expressiveness of programs that manipulate linear resources: with a single type system, programs can combine the advantages of the capability calculus and traditional linear types. Finally, by formalizing three particular encodings of aliasing within a linear type system, this paper helps dispel the common wisdom that linearity implies non-aliasing: the central message of this paper is that linear types *can* express aliasing of linear resources.

2 Aliasing with linear types

Sections 3-7 describe the complete encodings of the capability calculus. This section describes the intuition behind the encodings. After all, it seems counterintuitive at first that linear types, which prohibit a program from duplicating references to linear values, could express the aliasing of linear values. In fact, the ideas that underlie the encodings are fairly simple, and are easy to express using the *hello* and *goodbye* examples from section 1. The first step is to rewrite the examples using linear types:

$$\begin{array}{l}
\mathit{open} : \mathit{string} \rightarrow \mathit{file} \\
\mathit{close} : \mathit{file} \rightarrow () \\
\mathit{write} : \mathit{file} \otimes \mathit{string} \rightarrow \mathit{file} \\
\mathit{hello} : \mathit{file} \otimes \mathit{file} \rightarrow \mathit{file} \otimes \mathit{file} \\
\mathit{hello}\ \langle x_1 : \mathit{file}, x_2 : \mathit{file} \rangle = \langle \mathit{write}\ \langle x_1, \text{``}\mathit{hello}\text{''} \rangle, \mathit{write}\ \langle x_2, \text{``}\mathit{hello}\text{''} \rangle \rangle \\
\mathit{goodbye} : \mathit{file} \otimes \mathit{file} \rightarrow () \\
\mathit{goodbye}\ \langle x_1 : \mathit{file}, x_2 : \mathit{file} \rangle = \mathit{close}\ (\mathit{write}\ \langle x_1, \text{``}\mathit{bye}\text{''} \rangle);\ \mathit{close}\ (\mathit{write}\ \langle x_2, \text{``}\mathit{bye}\text{''} \rangle)
\end{array}$$

Nonlinear pairs (e_1, e_2), of type $\tau_1 \times \tau_2$, cannot contain linear values (otherwise, a program could duplicate a linear value by storing it in a nonlinear pair and duplicating the nonlinear pair). Therefore, the new versions of *write*, *hello*, and *goodbye* store files in linear pairs $\langle e_1, e_2 \rangle$ of linear type $\tau_1 \otimes \tau_2$. In addition, the new version of *write* returns the linear file that it receives as an argument; this allows the caller to continue using the file after writing to it. For example, the following well-typed function writes to a linear file twice, then closes it:

$$
\begin{aligned}
&\mathit{byebye} : \mathit{file} \rightarrow () \\
&\mathit{byebye}\ (x : \mathit{file}) = \\
&\quad \text{let}\, x' = \mathit{write}\ \langle x, \text{“}\mathit{bye}\text{”}\rangle \text{ in} \\
&\quad \text{let}\, x'' = \mathit{write}\ \langle x', \text{“}\mathit{bye}\text{”}\rangle \text{ in} \\
&\quad \mathit{close}\ x''
\end{aligned}
$$

A program cannot continue using a linear file after closing it, though, because *close* does not return its argument back to the caller[1].

The new version of *hello* can no longer be called with a single file as both arguments x_1 and x_2, as discussed in section 1. Section 2.1 rewrites *hello* to allow aliasing by using function abstraction, and section 2.2 rewrites *hello* to allow aliasing by using the linear choice operator.

2.1 Aliasing with function abstraction

The *hello* function takes two arguments x_1 and x_2, and it should be possible for these arguments to be two different files or for them to be the same file. Suppose that *hello* does not receive x_1 and x_2 directly, but instead accesses them by invoking methods on an object, whose private implementation might consist of two separate files or might consist of a single file. Let α be the private implementation of the object; an implementer could choose $\alpha = \mathit{file}$ in the case of a single file or $\alpha = \mathit{file} \otimes \mathit{file}$ in the case of separate files. Naively, the object's methods could be functions of type $\alpha \rightarrow \mathit{file}$, so that the method gets a file from the object's private implementation. If $\alpha = \mathit{file} \otimes \mathit{file}$, this type doesn't work; a function of type $\mathit{file} \otimes \mathit{file} \rightarrow \mathit{file}$ would have to discard one of the two files. A more appropriate type would be $\alpha \rightarrow \beta \otimes \mathit{file}$, where $\beta = ()$ if $\alpha = \mathit{file}$ and $\beta = \mathit{file}$ if $\alpha = \mathit{file} \otimes \mathit{file}$. Furthermore, after *hello* gets one file from α, it should be able to get the other file from α and then return α back to the caller. Therefore, there should be reverse methods of type $\beta \otimes \mathit{file} \rightarrow \alpha$. Altogether, the object consists of one private implementation α, a method $f_1 : \alpha \rightarrow \beta \otimes \mathit{file}$ to acquire the file x_1, a method $g_1 : \beta \otimes \mathit{file} \rightarrow \alpha$ to release x_1, a method $f_2 : \alpha \rightarrow \beta \otimes \mathit{file}$ to acquire x_2, and a method $g_2 : \beta \otimes \mathit{file} \rightarrow \alpha$ to release x_2:

$$
\begin{aligned}
&\mathit{hello} : \forall\alpha.\forall\beta.\alpha \otimes (\alpha \rightarrow \beta \otimes \mathit{file}) \otimes (\beta \otimes \mathit{file} \rightarrow \alpha) \\
&\qquad\qquad\qquad \otimes(\alpha \rightarrow \beta \otimes \mathit{file}) \otimes (\beta \otimes \mathit{file} \rightarrow \alpha) \rightarrow \alpha \\
&\mathit{hello}\ \langle a : \alpha,\ f_1 : \alpha \rightarrow \beta \otimes \mathit{file},\ g_1 : \beta \otimes \mathit{file} \rightarrow \alpha, \\
&\qquad\qquad f_2 : \alpha \rightarrow \beta \otimes \mathit{file},\ g_2 : \beta \otimes \mathit{file} \rightarrow \alpha\rangle = \\
&\quad \text{let}\, \langle b_1, x_1\rangle = f_1\ a \text{ in} \\
&\quad \text{let}\, a' = g_1\ \langle b_1, \mathit{write}\ \langle x_1, \text{“}\mathit{hello}\text{”}\rangle\rangle \text{ in} \\
&\quad \text{let}\, \langle b_2, x_2\rangle = f_2\ a' \text{ in} \\
&\quad \text{let}\, a'' = g_2\ \langle b_2, \mathit{write}\ \langle x_2, \text{“}\mathit{hello}\text{”}\rangle\rangle \text{ in}\, a''
\end{aligned}
$$

Since *hello* is polymorphic over all α and β, it works both when α consists of one file and when α consists of two files. Nevertheless, *hello*'s caller knows what α is, and *hello* returns a value of type α, so that the caller recovers the linearity of the file or files after *hello* returns; thus, *hello* is able to use x_1 and x_2 as if they were aliased, without the caller losing track of the aliasing forever. Furthermore, if a devious reimplementation of *hello*'s body tried to acquire and close x_1, it would have no way to subsequently acquire and close x_2, since acquiring x_1 consumes α, and α is needed to acquire x_2. This is both reassuring, since it should be impossible to close both x_1 and x_2 if x_1 and x_2 refer to the same file, and expected, since the underlying linear type system never allows a program to close a file twice, no matter how deviously and cleverly the program employs function abstraction. Finally, even though α, β, and file are linear types, the function types $\alpha \rightarrow \beta \otimes \mathit{file}$ and

[1] A note on notation: linear logic often defines $\tau_1 \rightarrow \tau_2$ to mean $(!\tau_1) \multimap \tau_2$, while this paper follows a different convention [27][17]: $\tau_1 \rightarrow \tau_2$ means $!(\tau_1 \multimap \tau_2)$; for example, a program can call the function $\mathit{close} : \mathit{file} \rightarrow ()$ many times, each time with a linear file argument.

$\beta \otimes \mathit{file} \to \alpha$ are nonlinear, so that *hello* may use f_1, g_1, f_2, and g_2 as many times as it likes, or never use them at all.

Invoking the functions f_1, g_1, f_2, and g_2 adds considerable syntactic overhead to *hello*'s implementation. It's possible to factor out some of this overhead. First, define a type abbreviation:

$$\tau_1 \leftrightarrow \tau_2 = (\tau_1 \to \tau_2) \times (\tau_2 \to \tau_1)$$

Then create a wrapper around the *write* function:

$$\begin{aligned}
&\mathit{writeAliased} : \forall\alpha.\forall\beta.\alpha \otimes (\alpha \leftrightarrow \beta \otimes \mathit{file}) \otimes \mathit{string} \to \alpha \\
&\mathit{writeAliased}\ \langle a : \alpha,\ \langle f : \alpha \to \beta \otimes \mathit{file},\ g : \beta \otimes \mathit{file} \to \alpha\rangle,\ s : \mathit{string}\rangle = \\
&\quad \mathrm{let}\ \langle b, x\rangle = f\ a\ \mathrm{in}\ g\ \langle b, \mathit{write}\ \langle x,\ s\rangle\rangle
\end{aligned}$$

Then hello simplifies to:

$$\begin{aligned}
&\mathit{hello}\ \langle a : \alpha,\ y_1 : \alpha \leftrightarrow \beta \otimes \mathit{file},\ y_2 : \alpha \leftrightarrow \beta \otimes \mathit{file}\rangle = \\
&\quad \mathrm{let}\ a' = \mathit{writeAliased}\ \langle a,\ y_1,\ \text{``}\mathit{hello}\text{''}\rangle\ \mathrm{in} \\
&\quad \mathrm{let}\ a'' = \mathit{writeAliased}\ \langle a',\ y_2,\ \text{``}\mathit{hello}\text{''}\rangle\ \mathrm{in}\ a''
\end{aligned}$$

2.1.1 Capabilities and proofs

In addition to their syntactic overhead, the function calls to f_1, g_1, f_2, and g_2 add run-time overhead to hello's implementation. A couple of well-known extensions to the type system can eliminate this overhead. The first extension [7] splits linear resources into a run-time handle, of nonlinear type "$\rho\,\mathrm{handle}$", and a compile-time capability, of linear type "$\rho\,\mathrm{cap}$", where the type variable ρ ensures that a handle is always used with the appropriate capability (a $\rho_1\,\mathrm{handle}$ is incompatible with a $\rho_2\,\mathrm{cap}$). A compiler erases the capabilities after type-checking, so that capabilities add no run-time space or time overhead. Each file operation requires both the capability and the handle for a file:

$$\begin{aligned}
&\mathit{open} : \mathit{string} \to \exists\rho.\rho\,\mathrm{cap} \otimes \rho\,\mathrm{handle} \\
&\mathit{close} : \forall\rho.\rho\,\mathrm{cap} \otimes \rho\,\mathrm{handle} \to () \\
&\mathit{write} : \forall\rho.\rho\,\mathrm{cap} \otimes \rho\,\mathrm{handle} \otimes \mathit{string} \to \rho\,\mathrm{cap} \\
&\mathit{hello} : \forall\rho_1.\forall\rho_2.\forall\alpha.\forall\beta_1.\forall\beta_2.\alpha \otimes (\alpha \to \beta_1 \otimes \rho_1\,\mathrm{cap}) \otimes (\beta_1 \otimes \rho_1\,\mathrm{cap} \to \alpha) \otimes \rho_1\,\mathrm{handle} \\
&\qquad\qquad \otimes (\alpha \to \beta_2 \otimes \rho_2\,\mathrm{cap}) \otimes (\beta_2 \otimes \rho_2\,\mathrm{cap} \to \alpha) \otimes \rho_2\,\mathrm{handle} \to \alpha \\
&\mathit{hello}\ \langle a : \alpha,\ f_1 : \alpha \to \beta_1 \otimes \rho_1\,\mathrm{cap},\ g_1 : \beta_1 \otimes \rho_1\,\mathrm{cap} \to \alpha,\ h_1 : \rho_1\,\mathrm{handle} \\
&\qquad f_2 : \alpha \to \beta_2 \otimes \rho_2\,\mathrm{cap},\ g_2 : \beta_2 \otimes \rho_2\,\mathrm{cap} \to \alpha,\ h_2 : \rho_2\,\mathrm{handle}\rangle = \\
&\quad \mathrm{let}\ \langle b_1, x_1\rangle = f_1\ a\ \mathrm{in} \\
&\quad \mathrm{let}\ a' = g_1\ \langle b_1, \mathit{write}\ \langle x_1, h_1, \text{``}\mathit{hello}\text{''}\rangle\rangle\ \mathrm{in} \\
&\quad \mathrm{let}\ \langle b_2, x_2\rangle = f_2\ a'\ \mathrm{in} \\
&\quad \mathrm{let}\ a'' = g_2\ \langle b_2, \mathit{write}\ \langle x_2, h_2, \text{``}\mathit{hello}\text{''}\rangle\rangle\ \mathrm{in}\ a''
\end{aligned}$$

Since the handles are nonlinear, *hello* may alias them freely; the function abstraction is only necessary for the linear capabilities. Since the capabilities $\rho_1\,\mathrm{cap}$ and $\rho_2\,\mathrm{cap}$ occupy no memory at run-time, the types α, β_1, and β_2 also occupy no memory. (The new code requires two variables β_1 and β_2 rather than a single β, because there are now two capability types $\rho_1\,\mathrm{cap}$ and $\rho_2\,\mathrm{cap}$ rather than a single file type file.) At run-time, the functions f_1, g_1, f_2, and g_2 do nothing; they consume empty arguments and produce empty results. There's no reason to actually call them. The second extension to the type system [6] formalizes this observation by making f_1, g_1, f_2, and g_2 functions in a proof language rather than a functions in a programming language (the proof functions perform no run-time computation, perform no I/O, and always terminate, so there's no reason to actually call them at run time; the compiler erases the proofs after type-checking, just as it erases the capabilities). Section 6 describes a particular proof language in detail, but for the moment, assume that there are types $\tau_1 \Rightarrow \tau_2$ for proof functions, so that hello has type:

$$\begin{aligned}
&\mathit{hello} : \forall\rho_1.\forall\rho_2.\forall\alpha.\forall\beta_1.\forall\beta_2.\alpha \otimes (\alpha \Rightarrow \beta_1 \otimes \rho_1\,\mathrm{cap}) \otimes (\beta_1 \otimes \rho_1\,\mathrm{cap} \Rightarrow \alpha) \otimes \rho_1\,\mathrm{handle} \\
&\qquad\qquad \otimes (\alpha \Rightarrow \beta_2 \otimes \rho_2\,\mathrm{cap}) \otimes (\beta_2 \otimes \rho_2\,\mathrm{cap} \Rightarrow \alpha) \otimes \rho_2\,\mathrm{handle} \to \alpha
\end{aligned}$$

2.2 Aliasing with choice

This section describes a second way to implement aliasing, based on previous work by Walker [30] and O'Hearn [14]. As a starting point, consider the wrapper function *writeAliased* from section 2.1, updated to use the proof functions and capabilities from section 2.1.1:

$$writeAliased : \forall\rho.\forall\alpha.\forall\beta.(\alpha \otimes (\alpha \Leftrightarrow \beta \otimes \rho\,\mathrm{cap})) \otimes \rho\,\mathrm{handle} \otimes string \rightarrow \alpha$$

Actually, the file system's primitive *write* function could simply use this type in the first place, eliminating the need for a wrapper:

$$write : \forall\rho.\forall\alpha.\forall\beta.(\alpha \otimes (\alpha \Leftrightarrow \beta \otimes \rho\,\mathrm{cap})) \otimes \rho\,\mathrm{handle} \otimes string \rightarrow \alpha$$

Furthermore, since *write* is a trusted built-in function (perhaps written in an unsafe language like C), it can relax its type slightly. The *write* function requires a proof of type $\alpha \Rightarrow \beta \otimes \rho\,\mathrm{cap}$ to know that $\rho\,\mathrm{cap}$ still exists, which proves that the file pointed to by $\rho\,\mathrm{handle}$ is still open. The caller relies on *write*'s return value of α to ensure that *write* doesn't close the file; the caller passes in a proof of type $\beta \otimes \rho\,\mathrm{cap} \Rightarrow \alpha$ to allow *write* to reconstruct α after using $\alpha \Rightarrow \beta \otimes \rho\,\mathrm{cap}$. A trusted implementation of *write* needn't bother with $\beta \otimes \rho\,\mathrm{cap} \Rightarrow \alpha$, though; it can simply claim to return α, and the caller trusts this claim. The relaxed type for *write* is:

$$write : \forall\rho.\forall\alpha.\forall\beta.(\alpha \otimes (\alpha \Rightarrow \beta \otimes \rho\,\mathrm{cap})) \otimes \rho\,\mathrm{handle} \otimes string \rightarrow \alpha$$

In the new version of *write*, the scope of β need not be so large; the following type works just as well:

$$write : \forall\rho.\forall\alpha.(\alpha \otimes (\alpha \Rightarrow (\exists\beta.\beta) \otimes \rho\,\mathrm{cap})) \otimes \rho\,\mathrm{handle} \otimes string \rightarrow \alpha$$

Abbreviating $\exists\beta.\beta$ as "true", the type for *write* becomes:

$$write : \forall\rho.\forall\alpha.(\alpha \otimes (\alpha \Rightarrow \mathrm{true} \otimes \rho\,\mathrm{cap})) \otimes \rho\,\mathrm{handle} \otimes string \rightarrow \alpha$$

This relaxed type breaks with section 2.1 in a fundamental way — whereas the proof language in section 2.1 was merely used to optimize away run-time calls to functions, the soundness of the new version of *write* depends on the lack of side effects in the implementation of $\alpha \Rightarrow \beta \otimes \rho\,\mathrm{cap}$. In other words, the following type would be unsound:

$$badWrite : \forall\rho.\forall\alpha.(\alpha \otimes (\alpha \rightarrow \mathrm{true} \otimes \rho\,\mathrm{cap})) \otimes \rho\,\mathrm{handle} \otimes string \rightarrow \alpha$$

Suppose a programmer passes a function f of type $\alpha \rightarrow \mathrm{true} \otimes \rho\,\mathrm{cap}$ to *badWrite*. If *badWrite* doesn't actually call f at run time, then f can subvert soundness using non-termination (f can produce a return type $\mathrm{true} \otimes \rho\,\mathrm{cap}$ by infinitely recursing on itself, regardless of what α is). If, on the other hand, *badWrite* does call f, then f can deallocate some linear resource present in α, making it unsound for *badWrite* to return α. Either way, *badWrite* is unsafe.

With the new version of *write*, the type for *hello* becomes:

$$\begin{aligned} hello : \forall\rho_1.\forall\rho_2.\forall\alpha.(\alpha \otimes (\alpha \Rightarrow \mathrm{true} \otimes \rho_1\,\mathrm{cap}) \\ \otimes(\alpha \Rightarrow \mathrm{true} \otimes \rho_2\,\mathrm{cap})) \otimes \rho_1\,\mathrm{handle} \otimes \rho_2\,\mathrm{handle} \rightarrow \alpha \end{aligned}$$

One way to think about *hello*'s type is that *hello* receives an α and a choice of how to use α: *hello* can keep α as-is, it can turn it into $\mathrm{true} \otimes \rho_1\,\mathrm{cap}$, or it can turn it into $\mathrm{true} \otimes \rho_2\,\mathrm{cap}$. This is reminiscent of the linear choice operator "&" in linear logic [28]: if you have a value of type $\tau_1 \& \tau_2$, then you can choose either τ_1 or τ_2, but not both (in contrast to the linear pair operator "$\otimes$", where $\tau_1 \otimes \tau_2$ gives you both τ_1 and τ_2). The correspondence between *hello*'s type and the linear choice operator suggests an alternate version of *hello*, using "&" rather than "$\Rightarrow$":

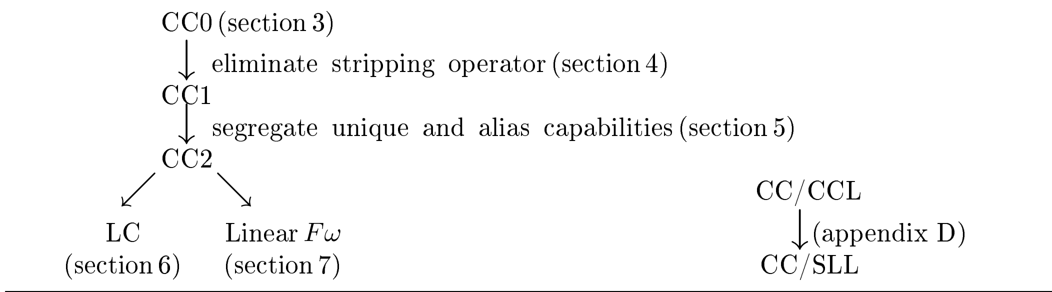

Figure 1: Translating CC0 to LC, CC0 to Linear $F\omega$, and CC/CCL to CC/SLL

$$hello' : \forall\rho_1.\forall\rho_2.\forall\gamma.(\gamma\&(\text{true} \otimes \rho_1\,\text{cap})\&(\text{true} \otimes \rho_2\,\text{cap})) \otimes \rho_1\,\text{handle} \otimes \rho_2\,\text{handle} \rightarrow \gamma$$

For example, if a caller has a pair of capabilities $\rho_1\,\text{cap} \otimes \rho_2\,\text{cap}$, then it can call $hello'$ with $\gamma = \rho_1\,\text{cap} \otimes \rho_2\,\text{cap}$, because:

$$(\rho_1\,\text{cap} \otimes \rho_2\,\text{cap}) \Rightarrow (\rho_1\,\text{cap} \otimes \rho_2\,\text{cap})\&(\text{true} \otimes \rho_1\,\text{cap})\&(\text{true} \otimes \rho_2\,\text{cap})$$

If, on the other hand, the caller has only a single capability $\rho\,\text{cap}$, it can choose $\gamma = \rho\,\text{cap}$, because:

$$(\rho\,\text{cap}) \Rightarrow (\rho\,\text{cap})\&(\text{true} \otimes \rho\,\text{cap})\&(\text{true} \otimes \rho\,\text{cap})$$

In fact, the types for *hello* and $hello'$ are interchangeable: *hello* can call $hello'$ by instantiating γ with α, and $hello'$ can call *hello* by instantiating α with $\gamma\&(\text{true} \otimes \rho_1\,\text{cap})\&(\text{true} \otimes \rho_2\,\text{cap})$. In section 6's encoding, "&" is strongly preferable to " $\Rightarrow$ ", because a linear propositional logic with only linear operators is decidable, while nonlinear operators like " $\Rightarrow$ " can destroy decidability [16].

2.3 Outline

Sections 2.1 and 2.2 demonstrated that linear types can express aliasing, at least in a simple example. The rest of the paper extends this expression to a complete language (see Figure 1). Section 3 introduces CC0, a slightly modified version of the calculus of capabilities [7]. Sections 6 and 7 present the translations of CC0 into two target languages. The first target language, called LC, retains CC0's syntax for expressions but replaces CC0's subcapability relation, bounded quantification, join operator, and stripping operator with a simple, decidable linear logic. To express aliasing, LC uses the linear choice operator, as described in section 2.2. The translation from CC0 to LC leaves the run-time behavior of a program unchanged: a CC0 program's type erasure is identical to the corresponding LC program's type erasure. LC is based on a language developed by David Walker [30], and the translation from CC0 to LC implements his suggested connection between linear types and the capability calculus (see [30], pp. 37-41).

The second target language, linear $F\omega$, is the higher-order polymorphic lambda calculus extended with linear types. Since linear $F\omega$ lacks LC's distinction between a programming language and a proof language, the translation uses the functional abstraction aliasing strategy described in section 2.1. One surprise in the CC0-to-linear $F\omega$ encoding is the reliance on type variables of kind Type $\rightarrow$ Type, which is not required in the CC0-to-LC encoding; the difference between the two encodings stems from the use of a proof language in LC but not in linear $F\omega$. It's an open question whether there is a CC0-to-linear $F2$ encoding (i.e. an encoding using only type variables of kind Type).

Although the CC0-to-linear $F\omega$ translation significantly changes all aspects of the source program (types and expressions), the changes made by the CC0-to-LC translation focus on

$$
\begin{array}{rrcl}
\textit{kinds} & \kappa & = & \mathrm{Type} \mid \mathrm{Res} \mid \mathrm{Cap} \\
\textit{constructors} & c & = & \alpha \mid \tau \mid C \\
\textit{ctor vars}\ \alpha, \beta, \epsilon, \rho, \ldots & & & \\
\textit{types} & \tau & = & \alpha \mid \rho\,\mathrm{handle} \mid \forall \alpha{:}\kappa.\tau \mid \forall \alpha \leq C.\tau \mid (C, \tau) \rightarrow 0 \mid \tau_1 \times \tau_2 \\
\textit{capabilities} & C & = & \epsilon \mid \emptyset \mid \{\rho^{\varphi}\} \mid C_1 \oplus C_2 \mid \overline{C} \\
\textit{multiplicities} & \varphi & = & 1 \mid + \\
\textit{ctor ctxts} & \Delta & = & \cdot \mid \Delta, \alpha : \kappa \mid \Delta, \epsilon \leq C \\
\textit{value ctxts} & \Gamma & = & \cdot \mid \Gamma, x : \tau \\
\textit{word values} & v & = & x \mid v[c : \kappa] \\
\textit{heap values} & h & = & \lambda \alpha{:}\kappa.h \mid \lambda \alpha \leq C.h \mid \lambda(C, x : \tau).e \mid (v_1, v_2) \\
\textit{declarations} & d & = & x = v \mid x = h \mid x = \#n\, v \mid \mathrm{new}\, \rho, x \mid \mathrm{free}\, v \mid \mathrm{use}\, v \\
\textit{expressions} & e & = & \mathrm{let}\, d\ \mathrm{in}\, e \mid v_1\ v_2 \mid \mathrm{halt}
\end{array}
$$

Figure 2: CC0 syntax

the logic of capabilities inside CC0's type system. In fact, it's easy to adapt this translation to Crary, Walker, and Morrisett's original calculus of capabilities (referred to here as "CC/CCL"), as shown in appendix D.

CC0 works with generic linear resources, assuming a few basic operations on resources. Section 7.1 replaces CC0's generic resources with a specific resource (heap objects) in order to implement alias types.

Sections 4 and 5 apply two preprocessing phases to the CC0 source program in preparation for the translations into LC and linear $F\omega$. The first preprocessing phase eliminates the stripping operator, which is a special type operator present in CC0 but not present in LC and linear Fω. The second preprocessing phase deals with CC0's flexible polymorphism over capabilities. From sections 2.1 and 2.2, it is clear that the translations to linear $F\omega$ and LC will treat the functions *hello* and *goodbye* differently. The challenge for the translation is that CC0 allows polymorphic functions that can behave both like *hello* and *goodbye*, depending on how their type arguments are instantiated. The second phase deals with this problem by splitting all capabilities into separate unique and alias parts.

The two preprocessing phases generate programs in a languages CC1 and CC2, which are variants of CC0. Unlike LC and linear $F\omega$, CC1 and CC2 are not designed for elegance and generality, and contain some ad-hoc features and restrictions that serve only to make the ultimate translations into LC and linear $F\omega$ easier. It would be possible, of course, to formulate the CC0-to-LC and CC0-to-linear $F\omega$ translations as single monolithic transformations, but this makes the translations less clear, and does not improve the quality of the generated LC code and linear $F\omega$ code.

3 CC0, a calculus of capabilities

As a starting point for the translations into LC and linear $F\omega$, this section describes CC0, a language based on the calculus of capabilities [7]. There are two major differences between CC0 and the calculus of capabilities:

- The calculus of capabilities supports a particular linear resource (regions), and annotates all heap value types with regions. This allows the calculus of capabilities to replace garbage collection with safe manual memory management. CC0, by contrast,

does not focus on any particular linear resource; it assumes some generic resource of kind "Res", with operations "new", "free", and "use", which correspond to "newrgn", "freergn", and hval/proj operations of the calculus of capabilities, and correspond to the "open", "close", and "write" operations in the examples from section 2. There are two reasons for not using regions in CC0. First, region annotations are orthogonal to the central topic of this paper (aliasing), and would obscure the translation's treatment of aliasing. Second, the successors to the calculus of capabilities [24][9] choose different linear resources, such as heap objects and sockets; there's no reason for this paper to prefer one resource over any other resource.

- The calculus of capabilities provided both a compile-time syntax for programs and a syntax for running programs. The latter includes the state of the heap, which is empty at compile time. Although the translations in this paper would apply to the run-time state as well as the compile-time state, the compile-time translation is of more practical interest, so for brevity's sake, CC0 omits the run-time state.

In addition, there are several minor differences, none of which are essential to the translation:

- CC0 deliberately omits recursive functions, in order to show that CC0's remaining features do not lead to non-termination. In particular, if the translation from CC0 to linear $F\omega$ preserves the run-time semantics of a program, then all well-typed CC0 programs terminate (because all well-typed linear $F\omega$ program terminate). The translation could easily accommodate recursion, assuming recursion is also added to the target languages.
- For brevity, CC0 omits the integer type.
- For simplicity, CC0 assumes each function takes exactly one argument, rather than multiple arguments.
- For clarity and notational consistency with LC and linear $F\omega$, CC0 uses pairs $\tau_1 \times \tau_2$ rather than n-tuples $\langle \tau_1, \ldots, \tau_n \rangle$.
- For clarity, CC0 breaks the $\forall[\Delta].(C, \tau) \rightarrow 0$ type into smaller primitives: $\forall \alpha : \kappa . \tau$ and $\forall \alpha \leq C . \tau$ and $(C, \tau) \rightarrow 0$.
- To simplify the presentation of the translations into CC1 and CC2, CC0 requires a kind annotation in the constructor application expression $v[c : \kappa]$.

Otherwise, CC0 is identical to the capability calculus. In particular, CC0 retains the syntax for capabilities, bounded polymorphism over capabilities, all of the capability equality rules, and all of the subcapability rules. Figure 2 shows the complete CC0 syntax, and appendix A contains CC0's complete static semantics. For more information about the capability calculus, see [7]; this section recaps the most important aspects and gives some short examples.

Consider the *goodbye* function from section 2. Figure 3 shows this function written in CC0 syntax. The first difference between the two versions is that CC0 functions are written in continuation-passing style (CPS). CPS functions do not return — they either halt the program or call another function. The *goodbye* function, for example, takes a continuation function k as an argument, and calls k when finished (much like a RISC assembly language program that jumps to an explicit return address upon completion).

The second difference between the two versions of *goodbye* is that section 2's version passed linear capabilities $\rho\,\mathrm{cap}$ as first-class arguments, while the CC0 version tracks capabilities in a special composite capability $C = \alpha \oplus \{\rho_1^1\} \oplus \{\rho_2^1\}$. In general, every CC0 function declares a capability in its type $(C, \tau) \rightarrow 0$. This capability is a precondition that callers of the function must satisfy. For example, *goodbye*'s capability specifies that in addition to some arbitrary α, which the caller chooses, the caller must provide capabilities for two

linear resources ρ_1 and ρ_2. Since *goodbye* deallocates both ρ_1 and ρ_2, *goodbye*'s continuation k cannot include ρ_1 and ρ_2 in its capability; the type system would prohibit *goodbye* from calling k if k had type $(\alpha \oplus \{\rho_1^1\} \oplus \{\rho_2^1\}, ()) \rightarrow 0$.

CC0 types refer to unaliased linear resources using "unique" capabilities, denoted by the syntax $\{\rho^1\}$. For unique capabilities, the CC0 join operator "$\oplus$" operator acts like the linear pair operator "$\otimes$". CC0 also supports aliased linear resources, as shown in figure 3's *hello* function. An alias capability $\{\rho^+\}$ indicates that ρ may appear elsewhere in the composite capability that contains $\{\rho^+\}$. For example, the capability $\alpha\oplus\{\rho_1^+\}\oplus\{\rho_2^+\}$ in *hello* indicates that ρ_1 and ρ_2 may occur in α, and that that ρ_1 and ρ_2 may be equal to each other. Since ρ_1 and ρ_2 are marked as potentially aliased, the type system prohibits *hello* from freeing ρ_1 and ρ_2 (although *hello* can still use ρ_1 and ρ_2) — this ensures that the free operation leaves no dangling capabilities to freed linear resources. A subcapability relation connects unique capabilities with alias capabilities; in particular, a unique capability is a subcapability of an alias capability: $\{\rho^1\} \leq \{\rho^+\}$.

For example, suppose that a function f contains capability $\{\rho_0^1\} \oplus \{\rho_1^1\} \oplus \{\rho_2^1\}$. Then f can call *hello* by choosing $\beta = \{\rho_0^1\} \oplus \{\rho_1^1\} \oplus \{\rho_2^1\}$ and $\alpha = \{\rho_0^1\}$. These choices satisfy *hello*'s subcapability bound $\beta \leq \alpha \oplus \{\rho_1^+\} \oplus \{\rho_2^+\}$:

$$\{\rho_0^1\} \oplus \{\rho_1^1\} \oplus \{\rho_2^1\} \leq \{\rho_0^1\} \oplus \{\rho_1^+\} \oplus \{\rho_2^+\}$$

Alternately, suppose that f only contains a single capability $\{\rho^1\}$. Then f can call *hello* by choosing $\rho_1 = \rho$ and $\rho_2 = \rho$, together with $\beta = \{\rho^1\}$ and $\alpha = \emptyset$. Alias capabilities are duplicable, so that $\{\rho^+\} = \{\rho^+\} \oplus \{\rho^+\}$. Therefore, these choices satisfy the subcapability bound $\beta \leq \alpha \oplus \{\rho_1^+\} \oplus \{\rho_2^+\}$:

$$\{\rho^1\} \leq \emptyset \oplus \{\rho^+\} \oplus \{\rho^+\}$$

Both versions of f can choose a continuation k that frees f's capabilities, since k takes β as its capability, and β consists of unique capabilities. Thus the calculus of capabilities allows *hello* to temporarily alias linear resources, without losing track of the linearity permanently.

4 From CC0 to CC1

The type variables α : Cap and β : Cap in section 3 demonstrate the capability calculus's polymorphism over capabilities. Since a program can instantiate type variables of kind Cap with both unique and alias capabilities, the type system must conservatively assume that these type variables are non-duplicable. To allow duplicable capability variables, the capability calculus introduces a stripping operator $\overline{C}$, which replaces all unique capabilities in C with alias capabilities (for example, $\overline{\{\rho^1\}} = \{\rho^+\}$). Stripped capabilities are duplicable, so that $\overline{\alpha} = \overline{\alpha} \oplus \overline{\alpha}$.

Traditional linear type systems contain no stripping operator. The "!" operator is similar, but not quite the same: stripping acts recursively on a capability, so that $\overline{C_1 \oplus C_2} = \overline{C_1} \oplus \overline{C_2}$, while $!(\tau_1 \times \tau_2)$ is not equal to $!\tau_1 \times !\tau_2$. Rather than adding the stripping operator to the target languages LC and linear $F\omega$, this section defines a translation $\mathcal{C}(C)$ that eliminates the stripping operator from capabilities in CC0:

$$\begin{aligned}
&\mathcal{C}(\alpha) = \alpha\\
&\mathcal{C}(\emptyset) = \emptyset\\
&\mathcal{C}(\{\alpha^\varphi\}) = \{\alpha^\varphi\}\\
&\mathcal{C}(C_1 \oplus C_2) = \mathcal{C}(C_1) \oplus \mathcal{C}(C_2)\\
&\mathcal{C}(\overline{C}) = \mathcal{S}(C)
\end{aligned}$$

The definition of $\mathcal{C}(C)$ is trivial except for the case of $\mathcal{C}(\overline{C})$, which requires an auxiliary definition $\mathcal{S}(C)$. For most cases, this auxiliary definition just follows CC0's equality rules for the stripping operator, which say that $\overline{\{\rho^1\}} = \{\rho^+\}$ and $\overline{C_1 \oplus C_2} = \overline{C_1} \oplus \overline{C_2}$ and $\overline{\overline{C}} = \overline{C}$ and $\overline{\emptyset} = \emptyset$:

$\tau_x = \tau_k \times (\rho_1\,\text{handle} \times \rho_2\,\text{handle})$
$\tau_k = (\alpha, ()) \to 0$
$\mathit{goodbye} : \forall\rho_1{:}\text{Res}.\forall\rho_2{:}\text{Res}.\forall\alpha{:}\text{Cap}.(\alpha \oplus \{\rho_1^1\} \oplus \{\rho_2^1\}, \tau_x) \to 0$
$\mathit{goodbye} = \lambda\rho_1{:}\text{Res}.\lambda\rho_2{:}\text{Res}.\lambda\alpha{:}\text{Cap}.\lambda(\alpha \oplus \{\rho_1^1\} \oplus \{\rho_2^1\}, x : \tau_x).$
$\quad \text{let}\,(k, (h_1, h_2)) = x \text{ in}$
$\quad \text{let use}\, h_1 \text{ in}$
$\quad \text{let free}\, h_1 \text{ in}$
$\quad \text{let use}\, h_2 \text{ in}$
$\quad \text{let free}\, h_2 \text{ in}\, k\ ()$

$\tau'_x = \tau'_k \times (\rho_1\,\text{handle} \times \rho_2\,\text{handle})$
$\tau'_k = (\beta, ()) \to 0$
$\mathit{hello} : \forall\rho_1{:}\text{Res}.\forall\rho_2{:}\text{Res}.\forall\alpha{:}\text{Cap}.\forall\beta \le \alpha \oplus \{\rho_1^+\} \oplus \{\rho_2^+\}.(\beta, \tau'_x) \to 0$
$\mathit{hello} = \lambda\rho_1{:}\text{Res}.\lambda\rho_2{:}\text{Res}.\lambda\alpha{:}\text{Cap}.\lambda\beta \le \alpha \oplus \{\rho_1^+\} \oplus \{\rho_2^+\}.\lambda(\beta, x : \tau'_x).$
$\quad \text{let}\,(k, (h_1, h_2)) = x \text{ in}$
$\quad \text{let use}\, h_1 \text{ in}$
$\quad \text{let use}\, h_2 \text{ in}\, k\ ()$

$\mathit{type\,abbreviation} :\ () = \forall\alpha{:}\text{Type}.(\emptyset, \alpha) \to 0$
$\mathit{expression\,abbreviation} :\ () = \lambda\alpha{:}\text{Type}.\lambda(\emptyset, x : \alpha).\text{halt}$
$\mathit{expression\,abbreviation} :\ (\text{let}\,(y, (z_1, z_2)) = x \text{ in } e)=$
$\quad (\text{let}\, y = \#1\, x \text{ in let}\, z = \#2\, x \text{ in let}\, z_1 = \#1\, z \text{ in let}\, z_2 = \#2\, z \text{ in}\, e)$

Figure 3: CC0 examples

$\mathcal{S}(\emptyset) = \emptyset$
$\mathcal{S}(\{\rho^\varphi\}) = \{\rho^+\}$
$\mathcal{S}(C_1 \oplus C_2) = \mathcal{S}(C_1) \oplus \mathcal{S}(C_2)$
$\mathcal{S}(\overline{C}) = \mathcal{S}(C)$

The only nontrivial case is $\mathcal{S}(\alpha)$, since there's no equality rule to simplify $\overline{\alpha}$. For this case, the translation invents a fresh type variable α_S for each α in the source program:

$\mathcal{S}(\alpha) = \alpha_S$

CC0's subcapability rules require that $C \le \overline{C}$ for all well-formed C. In particular, the translation must ensure that $\alpha \le \alpha_S$ in all contexts where α is a well-formed type of kind Cap. This means that any types or expressions declaring $\alpha : \text{Cap}$ must also declare α_S as a supertype of α. For example, in the translations $\mathcal{T}(\tau)$ of types below, a polymorphic source type $\forall\alpha{:}\text{Cap}.\tau$ turns into a target type that is polymorphic over both α and α_S, with the bound $\alpha \le \alpha_S$:

$\mathcal{T}(\alpha) = \alpha$
$\mathcal{T}(\rho\,\text{handle}) = \rho\,\text{handle}$
$\mathcal{T}((C, \tau) \to 0) = (\mathcal{C}(C), \mathcal{T}(\tau)) \to 0$
$\mathcal{T}(\tau_1 \times \tau_2) = \mathcal{T}(\tau_1) \times \mathcal{T}(\tau_2)$
$\mathcal{T}(\forall\alpha{:}\text{Type}.\tau) = \forall\alpha{:}\text{Type}.\mathcal{T}(\tau)$
$\mathcal{T}(\forall\rho{:}\text{Res}.\tau) = \forall\rho{:}\text{Res}.\mathcal{T}(\tau)$
$\mathcal{T}(\forall\alpha{:}\text{Cap}.\tau) = \forall\alpha_S{:}\text{Cap}^+.\forall\alpha{:}\text{Cap} \le \alpha_S.\mathcal{T}(\tau)$

(Note that α_S is only relevant to α of kind Cap, and does not affect the rules for $\mathcal{T}(\alpha)$, $\mathcal{T}(\rho\,\text{handle})$, $\mathcal{T}(\forall\alpha : \text{Type}.\tau)$, and $\mathcal{T}(\forall\rho : \text{Res}.\tau)$, which are only relevant to α : Type and ρ : Res.)

The definition of $\mathcal{T}(\forall\alpha \leq C.\tau)$ is similar to the definition of $\mathcal{T}(\forall\alpha:\mathrm{Cap}.\tau)$, in that α must be a subcapability of α_s, but also requires α to be a subcapability of $\mathcal{C}(C)$:

$$\mathcal{T}(\forall\alpha \leq C.\tau) = \forall\alpha_S:\mathrm{Cap}^+ \leq \mathcal{S}(C).\forall\alpha:\mathrm{Cap} \leq \mathcal{C}(C), \alpha_S.\mathcal{T}(\tau)$$

This requires the target language CC1 to support multiple bounds on capability variables. In addition, CC1 supports a new kind Cap^+ for capabilities that contain no unique capabilities $\{\rho^1\}$. Both of these new features persist into CC2, and are then eliminated in the translations to LC and linear $F\omega$.

The syntax of CC1's kinds, types, and capabilities is as follows:

$$\begin{array}{lrcl}
\textit{kinds} & \kappa & = & \mathrm{Type} \mid \mathrm{Res} \mid \mathrm{Cap} \mid \mathrm{Cap}^+ \\
\textit{types} & \tau & = & \alpha \mid \rho\,\mathrm{handle} \mid \forall\alpha:\kappa.\tau \mid (C,\tau) \to 0 \mid \tau_1 \times \tau_2 \\
 & & \mid & \forall\alpha:\mathrm{Cap}^+ \leq C.\tau \mid \forall\alpha:\mathrm{Cap} \leq C_0, C_1, \ldots, C_n.\tau \\
\textit{capabilities} & C & = & \epsilon \mid \emptyset \mid \{\rho^\varphi\} \mid C_1 \oplus C_2
\end{array}$$

Appendix A contains the complete syntax and rules for CC1, including additional rules and minor changes to the syntax for h and Δ. Appendix C contains the CC0-to-CC1 translations $\mathcal{E}(e)$ for expressions, $\mathcal{D}(d)$ for declarations, $\mathcal{V}(v)$ for values, $\mathcal{H}(h)$ for heap values, $\mathbf{\Delta}(\Delta)$ for type variable environments, and $\mathbf{\Gamma}(\Gamma)$ for variable environments, as well as lemmas for the type-correctness of these translations (the proofs appear in [12]).

A particularly important lemma is that type equality persists from CC0 to CC1, so that if $\Delta \vdash C_1 = C_2 : \mathrm{Cap}$ in CC0, then $\mathbf{\Delta}(\Delta) \vdash \mathcal{C}(C_1) = \mathcal{C}(C_2) : \mathrm{Cap}$ in CC1. The most challenging rule for this lemma is the CC0 rule for duplication of stripped capabilities:

$$\frac{\Delta \vdash C : \mathrm{Cap}}{\Delta \vdash \overline{C} = \overline{C} \oplus \overline{C} : \mathrm{Cap}}$$

Clearly this rule should not apply to unstripped capabilities, because unique capabilities $\{\rho^1\}$ should not be duplicable. What's not clear is how to write the rule without mentioning the stripping operator, which CC1 lacks. CC1's solution is to introduce a new kind Cap^+ for duplicable capabilities (i.e. capabilities containing no unique capabilities $\{\rho^1\}$). The kinding rules specify that alias capabilities $\{\rho^+\}$ have both kind Cap and Cap^+, but unique capabilities only have kind Cap:

$$\frac{\Delta \vdash \rho : \mathrm{Res}}{\Delta \vdash \{\rho^1\} : \mathrm{Cap}} \qquad \frac{\Delta \vdash \rho : \mathrm{Res}}{\Delta \vdash \{\rho^+\} : \mathrm{Cap}^+} \qquad \frac{\Delta \vdash C : \mathrm{Cap}^+}{\Delta \vdash C : \mathrm{Cap}}$$

Therefore, composite capabilities of kind Cap may contain both alias and unique capabilities, but composite capabilities of kind Cap^+ may contain only alias capabilities:

$$\frac{\Delta \vdash C_1 : \mathrm{Cap} \qquad \Delta \vdash C_2 : \mathrm{Cap}}{\Delta \vdash C_1 \oplus C_2 : \mathrm{Cap}} \qquad \frac{\Delta \vdash C_1 : \mathrm{Cap}^+ \qquad \Delta \vdash C_2 : \mathrm{Cap}^+}{\Delta \vdash C_1 \oplus C_2 : \mathrm{Cap}^+}$$

With these rules (and a few others, in appendix A), all stripped capabilities in a CC0 program turn into CC1 capabilities of kind Cap^+, which are duplicable via the following CC1 rule:

$$\frac{\Delta \vdash C : \mathrm{Cap}^+}{\Delta \vdash C = C \oplus C : \mathrm{Cap}^+}$$

5 From CC1 to CC2

The capability calculus's polymorphism over capabilities presents a challenge to LC and linear $F\omega$. Consider the following CC1 function type:

$$\forall\alpha_1{:}\mathrm{Cap}.\forall\alpha_2{:}\mathrm{Cap}.(\alpha_1 \oplus \alpha_2, (\alpha_1 \oplus \alpha_2, ()) \to 0) \to 0$$

If α_1 and α_2 represent unique capabilities $\{\rho_1^1\}$ and $\{\rho_2^1\}$, then $\alpha_1 \oplus \alpha_2$ should turn into a linear pair in LC and linear $F\omega$. On the other hand, if α_1 and α_2 represent alias capabilities $\{\rho_1^+\}$ and $\{\rho_2^+\}$, then the LC representation of $\alpha_1 \oplus \alpha_2$ would involve the choice operator (as described in section 2.2) and the linear $F\omega$ representation of $\alpha_1 \oplus \alpha_2$ would involve extra functions (as described in section 2.1). Since the function type shown above is polymorphic over all capabilities α_1 and α_2, the translated function type must handle both the unique case and the alias case.

To handle capability polymorphism, the CC1-to-CC2 translation splits every capability C into two parts: a unique part U and an alias part A, which combine to form the complete capability $C = U \boxplus A$, where $\boxplus$ is a pairing operator for joining unique and alias capabilities together. In particular, each capability variable α becomes two variables α_U and α_A; the former holds the unique part of α and the latter holds the alias part of α. The function type shown above translates to:

$$\begin{array}{l}\mathcal{T}(\forall\alpha_1{:}\mathrm{Cap}.\forall\alpha_2{:}\mathrm{Cap}.(\alpha_1 \oplus \alpha_2, (\alpha_1 \oplus \alpha_2, ()) \to 0) \to 0) = \\ \quad \forall\alpha_{1A}{:}\mathrm{Cap}^+.\forall\alpha_{1U}{:}\mathrm{Cap}^1.\forall\alpha_{2A}{:}\mathrm{Cap}^+.\forall\alpha_{2U}{:}\mathrm{Cap}^1. \\ \qquad ((\alpha_{1U} \oplus \alpha_{2U}) \boxplus (\alpha_{1A} \oplus \alpha_{2A}), ((\alpha_{1U} \oplus \alpha_{2U}) \boxplus (\alpha_{1A} \oplus \alpha_{2A}), \mathcal{T}(())) \to 0) \to 0\end{array}$$

In this form, the subsequent translations into LC and linear $F\omega$ can deal with the unique pair $\alpha_{1U} \oplus \alpha_{2U}$ separately from the alias pair $\alpha_{1A} \oplus \alpha_{2A}$, and yet callers of the function above can still instantiate $\alpha_1 = \alpha_{1U} \boxplus \alpha_{1A}$ and $\alpha_2 = \alpha_{2U} \boxplus \alpha_{2A}$ with arbitrary mixtures of unique and alias capabilities.

The syntax of CC2's kinds, constructors, types, and capabilities is as follows (the complete syntax and rules appear in appendix A):

$$\begin{array}{rrcl} \textit{kinds} & \kappa & = & \mathrm{Type} \mid \mathrm{Res} \mid \mathrm{Cap}^{\varphi} \\ \textit{constructors} & c & = & \alpha \mid \tau \mid Q \\ \textit{types} & \tau & = & \alpha \mid \rho\,\mathrm{handle} \mid \forall\alpha{:}\kappa.\tau \mid (C,\tau) \to 0 \mid \tau_1 \times \tau_2 \\ & & \mid & \forall\alpha{:}\mathrm{Cap}^+\,\mathrm{where}\,U_1 \boxplus \alpha \le A_1, \ldots, U_n \boxplus \alpha \le A_n.\tau \\ \textit{pure capabilities} & Q, A, U & = & \alpha \mid \emptyset \mid \{\rho^{\varphi}\} \mid Q_1 \oplus Q_2 \\ \textit{mixed capabilities} & C & = & U \boxplus A \end{array}$$

The letters Q, A, and U denote "pure capabilities", which are only well-kinded if they consist entirely of unique capabilities or entirely of alias capabilities. For example, in an environment where ρ_1 and ρ_2 have kind Res, $Q = \{\rho_1^1\} \oplus \{\rho_2^1\}$ has kind Cap^1 and $Q = \{\rho_1^+\} \oplus \{\rho_2^+\}$ has kind Cap^+, but $Q = \{\rho_1^1\} \oplus \{\rho_2^+\}$ is not well-kinded. By convention, U refers to pure capabilities of kind Cap^1, A refers to pure capabilities of kind Cap^+, and Q refers to pure capabilities of either kind. The letter C is still used, but only as an abbreviation for the "mixed capability" $U \boxplus A$. There is no kind for mixed capabilities (unlike CC0 and CC1, CC2 does not contain the kind Cap), but CC2's rules sometimes use the syntax "$\Delta \vdash U \boxplus A : \mathrm{Cap}$" as an abbreviation for "$\Delta \vdash U : \mathrm{Cap}^1$ and $\Delta \vdash A : \mathrm{Cap}^+$". Similarly, there are no equality rules for mixed capabilities, but CC2's rules sometimes use "$\Delta \vdash U \boxplus A = U' \boxplus A' : \mathrm{Cap}$" as an abbreviation for "$\Delta \vdash U = U' : \mathrm{Cap}^1$ and $\Delta \vdash A = A' : \mathrm{Cap}^+$". The subcapability relation does require some interaction between the unique and alias capabilities, as discussed in section 5.1, but otherwise U and A live in separate, isolated worlds.

The CC1-to-CC2 capability translation splits each capability C into its unique and alias parts:

$$
\begin{array}{ll}
\mathcal{C}(C) = \mathcal{U}(C) \boxplus \mathcal{A}(C) & \\
\mathcal{U}(\alpha) = \alpha_U & \mathcal{A}(\alpha) = \alpha_A \\
\mathcal{U}(\emptyset) = \emptyset & \mathcal{A}(\emptyset) = \emptyset \\
\mathcal{U}(\{\rho^1\}) = \{\rho^1\} & \mathcal{A}(\{\rho^1\}) = \emptyset \\
\mathcal{U}(\{\rho^+\}) = \emptyset & \mathcal{A}(\{\rho^+\}) = \{\rho^+\} \\
\mathcal{U}(C_1 \oplus C_2) = \mathcal{U}(C_1) \oplus \mathcal{U}(C_2) & \mathcal{A}(C_1 \oplus C_2) = \mathcal{A}(C_1) \oplus \mathcal{A}(C_2)
\end{array}
$$

Given the definition of $\mathcal{C}(C)$, the CC1-to-CC2 type translation is straightforward for types that do not involve subcapability bounds:

$$
\begin{array}{l}
\mathcal{T}(\alpha) = \alpha \\
\mathcal{T}(\rho\,\text{handle}) = \rho\,\text{handle} \\
\mathcal{T}((C, \tau) \to 0) = (\mathcal{C}(C), \mathcal{T}(\tau)) \to 0 \\
\mathcal{T}(\tau_1 \times \tau_2) = \mathcal{T}(\tau_1) \times \mathcal{T}(\tau_2) \\
\mathcal{T}(\forall\alpha{:}\text{Type}.\tau) = \forall\alpha{:}\text{Type}.\mathcal{T}(\tau) \\
\mathcal{T}(\forall\alpha{:}\text{Res}.\tau) = \forall\alpha{:}\text{Res}.\mathcal{T}(\tau) \\
\mathcal{T}(\forall\alpha{:}\text{Cap}.\tau) = \forall\alpha_A{:}\text{Cap}^+.\forall\alpha_U{:}\text{Cap}^1.\mathcal{T}(\tau) \\
\mathcal{T}(\forall\alpha{:}\text{Cap}^+.\tau) = \forall\alpha_A{:}\text{Cap}^+.[\alpha_U \leftarrow \emptyset]\mathcal{T}(\tau)
\end{array}
$$

Notice that the rule for $\forall\alpha : \text{Cap}^+.\tau$ substitutes $\emptyset$ for α_U rather than quantifying over all possible α_U. Any C substituted for $\alpha : \text{Cap}^+$ must have kind Cap^+, and for any C of kind Cap^+, $\mathcal{U}(C) = \emptyset$, so $\mathcal{T}(\forall\alpha{:}\text{Cap}^+.\tau)$ need not quantify over all α_U. The lemmas describing the type correctness of the CC1-to-CC2 translation (see appendix C) must account for the substitution of $\emptyset$ into α_U, so the translation defines a composite substitution $[\Delta]$:

$$
\begin{array}{l}
[\,\cdot\,] = [] \\
[\alpha : \text{Type}, \Delta] = [\Delta] \\
[\alpha : \text{Res}, \Delta] = [\Delta] \\
[\alpha : \text{Cap}, \Delta] = [\Delta] \\
[\alpha : \text{Cap}^+, \Delta] = [\alpha_U \leftarrow \emptyset]\,[\Delta]
\end{array}
$$

One important lemma is that the CC1-to-CC2 translation preserves capability equality: if $\vdash \Delta$ and $\Delta \vdash C_1 = C_2 : \kappa$ in CC1, where $\kappa = \text{Cap}$ or $\kappa = \text{Cap}^+$, then $\mathbf{\Delta}(\Delta) \vdash [\Delta]\mathcal{U}(C_1) = [\Delta]\mathcal{U}(C_2) : \text{Cap}^1$ and $\mathbf{\Delta}(\Delta) \vdash [\Delta]\mathcal{A}(C_1) = [\Delta]\mathcal{A}(C_2) : \text{Cap}^+$ in CC2.

5.1 Subcapabilities in CC2

CC2's unique and alias capabilities have separate translations $\mathcal{U}(U)$ and $\mathcal{A}(A)$, separate kinding judgments $\Delta \vdash U : \text{Cap}^1$ and $\Delta \vdash A : \text{Cap}^+$, and separate equality rules $\Delta \vdash U = U' : \text{Cap}^1$ and $\Delta \vdash A = A' : \text{Cap}^+$. The unique and alias parts cannot live in complete isolation, though, because the subcapability relationship $\{\rho^1\} \leq \{\rho^+\}$ lets capabilities migrate from the unique part to the alias part. Therefore, expecting all subcapability relations to conform to either $U \leq U'$ or $A \leq A'$ is too restrictive — the subcapability relation must at least allow relations of the form $U \leq A$ in order to capture $\{\rho^1\} \leq \{\rho^+\}$. The most obvious solution is allow fully general subcapability relations of the form $U \boxplus A \leq U' \boxplus A'$, and to adapt CC0's subcapability rules to CC2's syntax. Surprisingly, this general form is too permissive to support the CC2-to-LC and CC2-to-linear $F\omega$ translations. The problem stems from CC0's congruence rule for subcapabilities:

$$\frac{\Delta \vdash C_1 \leq C_1' \qquad \Delta \vdash C_2 \leq C_2'}{\Delta \vdash C_1 \oplus C_2 \leq C_1' \oplus C_2'}$$

The straightforward adaptation of this to CC2's syntax is:

$$\frac{\Delta \vdash U_1 \boxplus A_1 \leq U_1' \boxplus A_1' \qquad \Delta \vdash U_2 \boxplus A_2 \leq U_2' \boxplus A_2'}{\Delta \vdash (U_1 \oplus U_2) \boxplus (A_1 \oplus A_2) \leq (U_1' \oplus U_2') \boxplus (A_1' \oplus A_2')}$$

There's no reason to believe that this rule is unsound; after all, it can be derived in CC0's type system if $\boxplus$ is replaced by $\oplus$. Nevertheless, this rule is not very useful to the section 6's CC2-to-LC translation and section 7's CC2-to-linear $F\omega$ translation, because the corresponding judgment in LC would be incorrect (as would the corresponding judgment in linear $F\omega$):

$$\begin{array}{l} U_1 \otimes (A_1 \otimes \text{true}) \Rightarrow U_1' \otimes (A_1' \otimes \text{true}), \\ U_2 \otimes (A_2 \otimes \text{true}) \Rightarrow U_2' \otimes (A_2' \otimes \text{true}) \\ \vdash (U_1 \otimes U_2) \otimes (((A_1 \otimes \text{true}) \& (A_2 \otimes \text{true})) \otimes \text{true}) \Rightarrow \\ \quad (U_1' \otimes U_2') \otimes (((A_1' \otimes \text{true}) \& (A_2' \otimes \text{true})) \otimes \text{true}) \end{array}$$

The following instance of this judgment shows why the judgment is not correct:

$$\begin{array}{l} \emptyset \otimes (X \otimes \text{true}) \Rightarrow X \otimes (\emptyset \otimes \text{true}), \\ \emptyset \otimes (Y \otimes \text{true}) \Rightarrow Y \otimes (\emptyset \otimes \text{true}) \\ \vdash (\emptyset \otimes \emptyset) \otimes (((X \otimes \text{true}) \& (Y \otimes \text{true})) \otimes \text{true}) \Rightarrow \\ \quad (X \otimes Y) \otimes (((\emptyset \otimes \text{true}) \& (\emptyset \otimes \text{true})) \otimes \text{true}) \end{array}$$

The premises in this example are true, but the conclusion would require that $(X \otimes \text{true}) \& (Y \otimes \text{true})$ imply $X \otimes Y$, which is not correct: having a choice between X and Y does not give you both X and Y. The other direction is correct — $X \otimes Y$ does imply $(X \otimes \text{true}) \& (Y \otimes \text{true})$ — so clearly this example reversed something. The culprit is the choice of $A_1 = U_1' = X$ and $A_2 = U_2' = Y$. These choices assume that capabilities migrate from aliased parts to unique parts: $\emptyset \boxplus X \leq X \boxplus \emptyset$ and $\emptyset \boxplus Y \leq Y \boxplus \emptyset$. In other words, they assume that $\{\rho^+\} \leq \{\rho^1\}$, which is exactly backwards. CC0, CC1, and CC2 allow $\{\rho^1\} \leq \{\rho^+\}$ and prohibit $\{\rho^+\} \leq \{\rho^1\}$, which is why the congruence rule shown above works for CC2. By contrast, LC and linear $F\omega$ have no explicit $\{\rho^1\} \leq \{\rho^+\}$ rule, because they lack an explicit subcapability relation altogether.

Luckily, it is not difficult to modify CC2's subcapability relation to make the CC2-to-LC and CC2-to-linear $F\omega$ translations smoother. Although $U \boxplus A \leq U' \boxplus A'$ is too permissive, and segregated $U \leq U'$ and $A \leq A'$ are too restrictive, there's a middle ground that allows $\{\rho^1\} \leq \{\rho^+\}$ while still preventing $\{\rho^+\} \leq \{\rho^1\}$ from sneaking into LC and linear $F\omega$. If CC2 restricts subcapabilities to have the syntax $U \boxplus A \leq \emptyset \boxplus A'$, then it is *syntactically* impossible to express $\{\rho^+\} \leq \{\rho^1\}$ (assuming that U, A, and A' have the appropriate kinds). For example, this syntax forces the U_1' and U_2' to be $\emptyset$ in the LC judgment above, so that the judgment becomes:

$$\begin{array}{l} U_1 \otimes (A_1 \otimes \text{true}) \Rightarrow \emptyset \otimes (A_1' \otimes \text{true}), \\ U_2 \otimes (A_2 \otimes \text{true}) \Rightarrow \emptyset \otimes (A_2' \otimes \text{true}) \\ \vdash (U_1 \otimes U_2) \otimes (((A_1 \otimes \text{true}) \& (A_2 \otimes \text{true})) \otimes \text{true}) \Rightarrow \\ \quad (\emptyset \otimes \emptyset) \otimes (((A_1' \otimes \text{true}) \& (A_2' \otimes \text{true})) \otimes \text{true}) \end{array}$$

This judgment is correct. There are other possible syntactic restrictions (e.g. $U \boxplus \emptyset \leq U' \boxplus A'$ or $U \boxplus \emptyset \leq \emptyset \boxplus A'$), but $U \boxplus A \leq \emptyset \boxplus A'$ offers the cleanest way to encode the general case $U \boxplus A \leq U' \boxplus A'$:

$$(U \boxplus A \leq U' \boxplus A') \Leftrightarrow \exists U_B.(U = U_B \oplus U' \textit{ and } U_B \boxplus A \leq \emptyset \boxplus A')$$

This encoding allows capabilities to migrate from U to A' using the capability U_B as a conduit. For example, to express

$$(\{\rho_1^1\} \oplus \{\rho_2^1\}) \boxplus \{\rho_3^+\} \leq \{\rho_1^1\} \boxplus (\{\rho_2^+\} \oplus \{\rho_3^+\})$$

choose $U_B = \{\rho_2^1\}$:

$$\{\rho_1^1\} \oplus \{\rho_2^1\} = \{\rho_2^1\} \oplus \{\rho_1^1\} \textit{ and } \{\rho_2^1\} \boxplus \{\rho_3^+\} \le \emptyset \boxplus (\{\rho_2^+\} \oplus \{\rho_3^+\})$$

Furthermore, the encoding does *not* allow capabilities to migrate the wrong way (from A to U') — there is no way for the encoding to express $\emptyset \boxplus \{\rho^+\} \le \{\rho^1\} \boxplus \emptyset$, because there's no way to satisfy $U = U_B \oplus U'$ if $U = \emptyset$ and $U' = \{\rho^1\}$.

Appendix C defines rules for a relation $\Delta \vdash (C \le C') \rightsquigarrow U_B$ that generate an appropriate U_B in CC2 for any subcapability derivation $\Delta \vdash C \le C'$ in CC1. For example, the rules produce:

$$\Delta \vdash ((\{\rho_1^1\} \oplus \{\rho_2^1\}) \boxplus \{\rho_3^+\} \le \{\rho_1^1\} \boxplus (\{\rho_2^+\} \oplus \{\rho_3^+\})) \rightsquigarrow \{\rho_2^1\}$$

The key lemma for the subcapability relation states that if $\vdash \Delta$ and $\Delta \vdash (C \le C') \rightsquigarrow U_B$ in CC1, then $\mathbf{\Delta}(\Delta) \vdash [\Delta]\mathcal{U}(C) = [\Delta]U_B \oplus [\Delta]\mathcal{U}(C') : \text{Cap}^1$ and $\mathbf{\Delta}(\Delta) \vdash [\Delta]U_B \boxplus [\Delta]\mathcal{A}(C) \le \emptyset \boxplus [\Delta]\mathcal{A}(C')$ in CC2.

To emphasize the syntactic restriction on CC2's subcapability relation, the rest of the paper writes $U \boxplus A \le \emptyset \boxplus A'$ simply as $U \boxplus A \le A'$.

5.2 Bounded quantification in CC2

CC1 defines two forms of bounded quantification: $\forall\alpha : \text{Cap}^+ \le C.\tau$ and $\forall\alpha : \text{Cap} \le C_0, C_1, \ldots, C_n.\tau$. To make the translation to CC2 easier, CC1 restricts C and $C_1, \ldots, C_n$ to have kind Cap^+:

$$\frac{\Delta \vdash C : \text{Cap}^+ \qquad \Delta, \alpha : \text{Cap}^+ \le C \vdash \tau : \text{Type}}{\Delta \vdash \forall\alpha{:}\text{Cap}^+ \le C.\tau : \text{Type}}$$

$$\frac{\Delta \vdash C_0 : \text{Cap} \qquad \Delta \vdash C_1 : \text{Cap}^+ \qquad \ldots \qquad \Delta \vdash C_n : \text{Cap}^+ \qquad \Delta, \alpha : \text{Cap} \le (C_0, C_1, \ldots, C_n) \vdash \tau : \text{Type}}{\Delta \vdash \forall\alpha{:}\text{Cap} \le C_0, C_1, \ldots, C_n.\tau : \text{Type}}$$

For any C of kind Cap^+, $\mathcal{U}(C) = \emptyset$. This means, for example, that the translation of $\forall\alpha{:}\text{Cap}^+ \le C.\tau$ need not worry about establishing a U_B for $\mathcal{U}(\alpha)$ and $\mathcal{U}(C)$, because both $\mathcal{U}(\alpha)$ and $\mathcal{U}(C)$ are equal to $\emptyset$. The simplest syntax for the translation would be:

$$\mathcal{T}(\forall\alpha{:}\text{Cap}^+ \le C.\tau) = \forall\alpha_A{:}\text{Cap}^+ \le \mathcal{A}(C).[\alpha_U \leftarrow \emptyset]\mathcal{T}(\tau)$$

CC2 actually uses a more general syntax for bounded quantification, allowing bounds of the form $U \boxplus \alpha_A \le A'$ on the variable α_A, rather than just $\alpha_A \le A'$:

$$\mathcal{T}(\forall\alpha{:}\text{Cap}^+ \le C.\tau) = \forall\alpha_A{:}\text{Cap}^+ \text{ where } \emptyset \boxplus \alpha_A \le \mathcal{A}(C).[\alpha_U \leftarrow \emptyset]\mathcal{T}(\tau)$$

The more general syntax allows a translation of $\forall\alpha{:}\text{Cap} \le C_0, C_1, \ldots, C_n.\tau$, where α and C_0 may have non-empty unique parts due to their kind Cap. This case requires encoding $\alpha \le C_0$ as described in section 5.1:

$$\mathcal{U}(\alpha) = U_B \oplus \mathcal{U}(C_0) \textit{ and } U_B \boxplus \mathcal{A}(\alpha) \le \mathcal{A}(C_0)$$

By definition, $\mathcal{U}(\alpha) = \alpha_U$ and $\mathcal{A}(\alpha) = \alpha_A$. Since these are variables, U_B must also be a variable:

$$\alpha_U = \alpha_B \oplus \mathcal{U}(C_0) \textit{ and } \alpha_B \boxplus \alpha_A \le \mathcal{A}(C_0)$$

Because $C_1 \ldots C_n$ all have kind Cap^+, each of the $\alpha \le C_1 \ldots \alpha \le C_n$ bounds already conforms to CC2's subcapability syntax ($U \boxplus A \le A'$) and needs no further encoding:

$$\alpha_U \boxplus \alpha_A \le \mathcal{A}(C_1) \qquad \ldots \qquad \alpha_U \boxplus \alpha_A \le \mathcal{A}(C_n)$$

All told, there is one equality bound ($\alpha_U = \alpha_B \oplus \mathcal{U}(C_0)$), one subcapability bound for C_0, and n subcapability bounds for $C_1 \ldots C_n$. The easiest way to handle the equality bound is to substitute $\alpha_B \oplus \mathcal{U}(C_0)$ for α_U. This changes each of the n subcapability bounds to:

$$(\alpha_B \oplus \mathcal{U}(C_0)) \boxplus \alpha_A \leq \mathcal{A}(C_1) \qquad \ldots \qquad (\alpha_B \oplus \mathcal{U}(C_0)) \boxplus \alpha_A \leq \mathcal{A}(C_n)$$

and yields a complete (and admittedly, complicated) translation of $\forall\alpha\!:\!\mathrm{Cap} \leq C_0, C_1, \ldots, C_n.\tau$:

$$\begin{array}{l} \mathcal{T}(\forall\alpha\!:\!\mathrm{Cap} \leq C_0, C_1, \ldots, C_n.\tau) = \\ \quad \forall\alpha_B\!:\!\mathrm{Cap}^1.\forall\alpha_A\!:\!\mathrm{Cap}^+ \text{ where } \quad \alpha_B \boxplus \alpha_A \leq \mathcal{A}(C_0), \\ \qquad\qquad (\alpha_B \oplus \mathcal{U}(C_0)) \boxplus \alpha_A \leq \mathcal{A}(C_1), \\ \qquad\qquad \vdots \\ \qquad\qquad (\alpha_B \oplus \mathcal{U}(C_0)) \boxplus \alpha_A \leq \mathcal{A}(C_n).[\alpha_U \leftarrow (\alpha_B \oplus \mathcal{U}(C_0))]\mathcal{T}(\tau) \end{array}$$

Any CC1 expression that applies a value v of type $\forall\alpha : \mathrm{Cap} \leq C_0, C_1, \ldots, C_n.\tau$ to some capability argument C must, when translated into CC2, find an appropriate U_B to plug in for α_B. Luckily, section 5.1 already established a relation $\Delta \vdash (C \leq C') \rightsquigarrow U_B$ for choosing U_B, and this relation guides the translation of a CC1 value $v[C : \mathrm{Cap}]$ into a CC2 value $\mathcal{V}(v[C : \mathrm{Cap}])$, where the translation is written as an annotation of v's typing judgment $\Delta; \Gamma \vdash v[C : \mathrm{Cap}] : \tau \rightsquigarrow \mathcal{V}(v[C : \mathrm{Cap}])$:

$$\frac{\begin{array}{c} \Delta; \Gamma \vdash v : \forall\alpha\!:\!\mathrm{Cap} \leq C_0, C_1, \ldots, C_n.\tau \rightsquigarrow v' \\ \Delta \vdash C \leq C_0 \rightsquigarrow U_B \\ \Delta \vdash C \leq C_1 \qquad \Delta \vdash C \leq C_n \\ \Delta \vdash C : \mathrm{Cap} \end{array}}{\Delta; \Gamma \vdash v[C : \mathrm{Cap}] : [\alpha \leftarrow C]\tau \rightsquigarrow v'[U_B : \mathrm{Cap}^1][\mathcal{A}(C) : \mathrm{Cap}^+]}$$

Note that CC2 could generalize its bounded quantification even further, allowing bounds $U \boxplus A \leq A'$ instead of just $U \boxplus \alpha \leq A'$, but would complicate the subsequent translations into LC and linear $F\omega$ (in particular, the translation into LC relies on substitution for α). CC1 and CC2 must to be general enough to handle all well-typed CC0 programs, but restrictive enough to support the final translations into LC and linear $F\omega$. The path through CC1 and CC2 is a thin and not-entirely-straight line.

6 From CC2 to LC

Figure 4 shows the syntax of LC, which is based on a language developed by Walker [30]. LC is identical to CC0 except that:

- LC lacks a subcapability relation (and therefore requires no bounded quantification).
- LC lacks a stripping operator.
- LC lacks multiplicity flags (it has $\{\rho\}$ rather than $\{\rho^1\}$ and $\{\rho^+\}$).
- LC uses standard linear logic operators $\otimes$, $\&$, and $\multimap$ in place of CC0's nonstandard join operator ($\oplus$).

Appendix A presents the complete typing rules for LC. Figure 5 shows the the linear sequent logic rules that govern capabilities. In each rule, each of the premises is strictly smaller than the conclusion $\Lambda \vdash C$, so that the height of any derivation of $\Lambda \vdash C$ is bounded by a function of the size of $\Lambda \vdash C$. This ensures that there is at least one algorithm for deciding the validity of C under assumptions $\Lambda = C_1, ..., C_n$: simply try all possible derivations with $\Lambda \vdash C$ as the conclusion [16]. Note that although the order of assumptions $\Lambda = C_1, ..., C_n$ is irrelevant

$$
\begin{array}{rrcl}
\textit{kinds} & \kappa & = & \text{Type} \mid \text{Res} \mid \text{Cap} \\
\textit{constructors} & c & = & \alpha \mid \tau \mid C \\
\textit{ctor vars}\ \alpha, \beta, \epsilon, \rho, \ldots & & & \\
\textit{types} & \tau & = & \alpha \mid \rho\,\text{handle} \mid \forall\alpha{:}\kappa.\tau \mid (C,\tau) \to 0 \mid \tau_1 \times \tau_2 \\
\textit{capabilities} & C & = & \epsilon \mid \emptyset \mid \{\rho\} \mid C_1 \otimes C_2 \mid C_1 \& C_2 \mid C_1 \multimap C_2 \mid \text{true} \\
\textit{cap ctxts} & \Lambda & = & C_1, \ldots, C_n \\
\textit{ctor ctxts} & \Delta & = & \cdot \mid \Delta, \alpha : \kappa \\
\textit{value ctxts} & \Gamma & = & \cdot \mid \Gamma, x : \tau \\
\textit{word values} & v & = & x \mid v[c : \kappa] \\
\textit{heap values} & h & = & \lambda\alpha{:}\kappa.h \mid \lambda(C, x : \tau).e \mid (v_1, v_2) \\
\textit{declarations} & d & = & x = v \mid x = h \mid x = \#n\,v \mid \text{new}\,\rho, x \mid \text{free}\,v \mid \text{use}\,v \\
\textit{expressions} & e & = & \text{let}\,d\,\text{in}\,e \mid v_1\ v_2 \mid \text{halt}
\end{array}
$$

Figure 4: LC Syntax

$$
C \vdash C \qquad\qquad \vdash \emptyset \qquad\qquad \frac{\Lambda \vdash C}{\Lambda, \emptyset \vdash C} \qquad\qquad \Lambda \vdash \text{true}
$$

$$
\frac{\Lambda_1 \vdash C_1 \qquad \Lambda_2 \vdash C_2}{\Lambda_1, \Lambda_2 \vdash C_1 \otimes C_2} \qquad\qquad \frac{\Lambda \vdash C_1 \qquad \Lambda \vdash C_2}{\Lambda \vdash C_1 \& C_2} \qquad\qquad \frac{\Lambda, C_1 \vdash C_2}{\Lambda \vdash C_1 \multimap C_2}
$$

$$
\frac{\Lambda, C_1, C_2 \vdash C_3}{\Lambda, C_1 \otimes C_2 \vdash C_3} \qquad\qquad \frac{\Lambda, C_k \vdash C_3}{\Lambda, C_1 \& C_2 \vdash C_3}(k \in \{1,2\}) \qquad\qquad \frac{\Lambda_1 \vdash C_1 \qquad \Lambda_2, C_2 \vdash C_3}{\Lambda_1, \Lambda_2, C_1 \multimap C_2 \vdash C_3}
$$

Figure 5: Linear sequent logic (without "!")

(C_1, C_2 is equivalent to C_2, C_1), derivations cannot duplicate or discard assumptions: C, C is not equivalent to C or C, C, C.

LC eliminates all of CC0's capability equality rules and subcapability rules, relying entirely on logical judgments $\Lambda \vdash C$ instead. LC defines capability equality in terms of logic judgments:

$$\frac{\Delta \vdash C_1 : \kappa \qquad \Delta \vdash C_2 : \kappa \qquad C_1 \vdash C_2 \qquad C_2 \vdash C_1}{\Delta \vdash C_1 = C_2 : \kappa}$$

In place of CC0's subcapability judgment $\Delta \vdash C_1 \leq C_2$, LC uses $C_1 \vdash C_2$. For example, the CC0 rule for use

$$\frac{\Delta; \Gamma \vdash v : \alpha\,\mathrm{handle} \qquad \Delta \vdash C \leq C' \oplus \{\alpha^+\}}{\Delta; \Gamma; C \vdash \mathrm{use}\, v \Longrightarrow \Delta; \Gamma; C}$$

becomes the following in LC:

$$\frac{\Delta; \Gamma \vdash v : \alpha\,\mathrm{handle} \qquad C \vdash \{\alpha\} \otimes \mathrm{true}}{\Delta; \Gamma; C \vdash \mathrm{use}\, v \Longrightarrow \Delta; \Gamma; C}$$

Translating unique CC2 capabilities U into LC is easy — just replace CC2's join operator $\oplus$ with LC's linear pair operator $\otimes$:

$\mathcal{U}(\alpha) = \alpha$
$\mathcal{U}(\emptyset) = \emptyset$
$\mathcal{U}(\{\rho^\varphi\}) = \{\rho\}$
$\mathcal{U}(U_1 \oplus U_2) = \mathcal{U}(U_1) \otimes \mathcal{U}(U_2)$

For alias capabilities A, the translation adopts the linear choice operator &, as described in section 2.2:

$\mathcal{A}(\alpha) = \alpha$
$\mathcal{A}(\emptyset) = \emptyset$
$\mathcal{A}(\{\rho^\varphi\}) = \{\rho\}$
$\mathcal{A}(A_1 \oplus A_2) = (\mathcal{A}(A_1) \otimes \mathrm{true}) \& (\mathcal{A}(A_2) \otimes \mathrm{true})$

To express the CC0 subcapability judgment $\{\rho_1^1\} \oplus \{\rho_2^1\} \leq \{\rho_1^+\} \oplus \{\rho_1^+\}$, the translation ensures that $\mathcal{U}(\{\rho_1^1\} \oplus \{\rho_2^1\}) \vdash \mathcal{A}(\{\rho_1^+\} \oplus \{\rho_1^+\})$:

$$\frac{\dfrac{\dfrac{\{\rho_1\} \vdash \{\rho_1\} \qquad \{\rho_2\} \vdash \mathrm{true}}{\{\rho_1\}, \{\rho_2\} \vdash \{\rho_1\} \otimes \mathrm{true}}}{\{\rho_1\} \otimes \{\rho_2\} \vdash \{\rho_1\} \otimes \mathrm{true}} \qquad \dfrac{\dfrac{\{\rho_2\} \vdash \{\rho_2\} \qquad \{\rho_1\} \vdash \mathrm{true}}{\{\rho_1\}, \{\rho_2\} \vdash \{\rho_2\} \otimes \mathrm{true}}}{\{\rho_1\} \otimes \{\rho_2\} \vdash \{\rho_1\} \otimes \mathrm{true}}}{\{\rho_1\} \otimes \{\rho_2\} \vdash (\{\rho_1\} \otimes \mathrm{true}) \& (\{\rho_2\} \otimes \mathrm{true})}$$

Note that the the $\otimes$true is required for the derivation to absorb the excess linear assumptions from the assumption list $\{\rho_1\}, \{\rho_2\}$, since a derivation cannot simply discard unwanted assumptions. Indeed, there is no derivation of $\{\rho_1\} \otimes \{\rho_2\} \vdash \{\rho_1\} \& \{\rho_2\}$. In general, the CC2 judgment $U \boxplus A \leq A'$ turns into an LC judgment:

$\mathcal{U}(U) \otimes \mathcal{A}(A) \vdash \mathcal{A}(A') \otimes \mathrm{true}$

(The $\otimes$true after the $\mathcal{A}(A')$ is necessary for relations like $\emptyset \boxplus \{\rho^+\} \oplus \{\rho^+\} \leq \{\rho^+\}$.)

Translation of types is straightforward, except for bounded quantification:

$\mathcal{T}(\alpha) = \alpha$
$\mathcal{T}(\rho\,\text{handle}) = \rho\,\text{handle}$
$\mathcal{T}((U \boxplus A, \tau) \to 0) = (\mathcal{U}(U) \otimes (\mathcal{A}(A) \otimes \text{true}), \mathcal{T}(\tau)) \to 0$
$\mathcal{T}(\tau_1 \times \tau_2) = \mathcal{T}(\tau_1) \times \mathcal{T}(\tau_2)$
$\mathcal{T}(\forall\alpha\!:\!\text{Type}.\tau) = \forall\alpha\!:\!\text{Type}.\mathcal{T}(\tau)$
$\mathcal{T}(\forall\alpha\!:\!\text{Res}.\tau) = \forall\alpha\!:\!\text{Res}.\mathcal{T}(\tau)$
$\mathcal{T}(\forall\alpha\!:\!\text{Cap}^{\varphi}.\tau) = \forall\alpha\!:\!\text{Cap}.\mathcal{T}(\tau)$

The translation of bounded quantification borrows from two alternate encodings of $\forall\alpha \leq \tau'.\tau$ [2][20][5]:

$(\forall\alpha \leq \tau'.\tau) = \forall\alpha.(\alpha \to \tau') \to \tau$
$(\forall\alpha \leq \tau'.\tau) = \forall\alpha.[\alpha \leftarrow (\alpha \wedge \tau')]\tau$

The first encoding uses a coercion function $(\alpha \to \tau')$ to make explicit the idea that any subtype can be coerced to a supertype. This encoding suggests expressing $\forall\alpha \leq C.\tau$ using a coercion implication $(\alpha \multimap C)$. The linearity of $(\alpha \multimap C)$ causes problems for the encoding (a nonlinear implication $!(\alpha \multimap C)$ would be preferable), but the type $(\alpha \multimap C)$ still plays a part in the translation described below.

The second encoding uses an intersection type $\tau_1 \wedge \tau_2$, which indicates a value that can be coerced to type τ_1 or τ_2; as in the first encoding, every value of type α in τ can be coerced to type τ', since $(\alpha \wedge \tau')$ can be coerced to type τ'. Walker [30] observed that since $\wedge$ corresponds to linear logic's & operator, $\forall\alpha \leq C.\tau$ can be encoded as $\forall\alpha.[\alpha \leftarrow (\alpha \& C)]\tau$. CC2's bounded quantification isn't as simple as $\alpha \leq C$, though — the bounds have the form $U \boxplus \alpha \leq A$. Consider again the first encoding, which would turn $U \boxplus \alpha \leq A$ into a coercion $\mathcal{U}(U) \otimes \alpha \multimap \mathcal{A}(A) \otimes \text{true}$. Now change the order of the argument pair to form $\alpha \otimes \mathcal{U}(U) \multimap \mathcal{A}(A) \otimes \text{true}$, and then *curry* the implication to form $\alpha \multimap (\mathcal{U}(U) \multimap \mathcal{A}(A) \otimes \text{true})$. This version of the coercion suggests an application of the second encoding to the bound $\alpha \leq (\mathcal{U}(U) \multimap \mathcal{A}(A) \otimes \text{true})$, which results in a type $\forall\alpha.[\alpha \leftarrow (\alpha \& (\mathcal{U}(U) \multimap \mathcal{A}(A) \otimes \text{true}))]\tau$. More precisely, the encoding of CC2's bounded quantification type is:

$$\mathcal{T}(\forall\alpha\!:\!\text{Cap}^{+}\,\text{where}\,U_1 \boxplus \alpha \leq A_1, \ldots, U_n \boxplus \alpha \leq A_n.\tau) =$$
$$\forall\alpha\!:\!\text{Cap}.[\alpha \leftarrow (\alpha \& (\mathcal{U}(U_1) \multimap \mathcal{A}(A_1) \otimes \text{true}) \& \ldots \& (\mathcal{U}(U_n) \multimap \mathcal{A}(A_n) \otimes \text{true}))]\mathcal{T}(\tau)$$

6.1 Incomplete collection

The capability calculus's most important property is its ability to safely track linear resource state changes, such as resource deallocation. The translation from CC0 to LC maintains this property. The capability calculus also guarantees a further property, known as *complete collection*: when a program halts, the typing rule for the halt expression guarantees an empty heap:

$$\frac{\Delta \vdash C = \emptyset : \text{Cap}}{\Delta; \Gamma; C \vdash \text{halt}}$$

Because the program can only terminate via the halt expression, this implies that programs can only terminate with an empty heap. Unfortunately, although the CC0-to-CC1 and CC1-to-CC2 translations preserve complete collection, the CC2-to-LC translation fails to maintain this property, and LC must relax halt's typing rule to accommodate the translation:

$$\Delta; \Gamma; C \vdash \text{halt}$$

Consider, for example, the translation $\mathcal{A}(\emptyset \oplus \emptyset) = (\emptyset \otimes \text{true}) \& (\emptyset \otimes \text{true})$. Because of the $\otimes$true that appears in each branch of the choice, $(\emptyset \otimes \text{true}) \& (\emptyset \otimes \text{true})$ is not equivalent to $\emptyset$ (under LC's rules, $\emptyset$ implies true, but true does not imply $\emptyset$). The following CC2 function accepts an $A = \emptyset \oplus \emptyset$ and then halts with an empty heap:

$\lambda(\emptyset \boxplus (\emptyset \oplus \emptyset), x : ()).\text{halt}$

The corresponding LC function cannot guarantee an empty heap.

It's important to note that the CC0-to-LC translation does not introduce any extra run-time garbage — if the CC0 program halts with an empty heap, then the corresponding LC program will also halt with an empty heap (because the LC program performs exactly the same sequence of allocations and deallocations as the CC0 program). The problem is LC's type system doesn't guarantee complete collection for all programs — even though programs translated from CC0 will halt with an empty heap, there are other well-typed LC programs that may halt with a non-empty heap.

There are four strategies for dealing with LC's incomplete collection:

- Ignore incomplete collection. Most practical systems must have a way to stop non-terminating programs or excessively long-running programs, and this requires that the system have a mechanism for reclaiming memory from programs that do not voluntarily halt. If this mechanism is already in place anyway, the system can also use it to clean up programs that do voluntarily halt.
- Embrace incomplete collection. Without the requirement that programs halt with an empty heap, the type system can make the subcapability more flexible, by adding a rule $\Delta \vdash C \leq \emptyset$. Alias types, for example, includes the rule $\Delta \vdash C \leq \emptyset$ and abandons complete collection [23].
- Recover complete collection by restricting the source language. For example, if CC2's function type $(U \boxplus A, \tau) \to 0$ is restricted to $(U \boxplus \emptyset, \tau) \to 0$, then the translation into LC no longer has to worry about $A = \emptyset \oplus \emptyset$ introducing spurious $\otimes$true capabilities into the environment. It's not obvious, though, how to push this restriction on CC2 back into CC1 and CC0.
- Recover complete collection by making the translation more precise. Section 6.2 discusses this option. Unfortunately, section 6.2's solution adds nonlinear capabilities to LC's logic, which probably makes the logic undecidable [16] unless the program contains additional annotations to guide the decision procedure.

6.2 Complete collection

The translation into LC forfeited complete collection because $\mathcal{A}(A_1 \oplus A_2) = (A_1 \otimes \text{true}) \& (A_2 \otimes \text{true})$ sloppily introduces $\otimes$true even when A_1 and A_2 are equivalent to $\emptyset$. The following revised translation eliminates this sloppiness by replacing $\otimes$true with $\otimes \mathcal{Z}(A)$:

$$\begin{aligned}
&\mathcal{A}(\alpha) = \alpha \\
&\mathcal{A}(\emptyset) = \emptyset \\
&\mathcal{A}(\{\rho^{\varphi}\}) = \{\rho\} \\
&\mathcal{A}(A_1 \oplus A_2) = (\mathcal{A}(A_1) \otimes \mathcal{Z}(A_1 \oplus A_2)) \& (\mathcal{A}(A_2) \otimes \mathcal{Z}(A_1 \oplus A_2))
\end{aligned}$$

where $\mathcal{Z}(A)$ is defined to be $\emptyset$ for A that are equivalent to $\emptyset$, and true otherwise:

$$\begin{aligned}
&\mathcal{Z}(\alpha) = \alpha_Z \\
&\mathcal{Z}(\emptyset) = \emptyset \\
&\mathcal{Z}(\{\rho^{\varphi}\}) = \text{true} \\
&\mathcal{Z}(A_1 \oplus A_2) = \mathcal{Z}(A_1) \otimes \mathcal{Z}(A_2)
\end{aligned}$$

Under the revised definition, $\mathcal{A}(\emptyset \oplus \emptyset) = (\emptyset \otimes (\emptyset \otimes \emptyset)) \& (\emptyset \otimes (\emptyset \otimes \emptyset))$, which is equivalent to $\emptyset$. More subtly, $\mathcal{A}(\alpha_1 \oplus \alpha_2)$ will be equivalent to $\emptyset$ if both α_1 and α_2 are instantiated with $\emptyset$ (or with any A equivalent to $\emptyset$). This requires a correlation between α and α_Z: whenever α is instantiated with $\mathcal{A}(A)$, α_Z should be instantiated with $\mathcal{Z}(A)$. To capture the relation between α and α_Z, the translation embeds the following invariant in the target program:

$$
\begin{array}{rcl}
\textit{kinds} & \kappa & = & \text{Type} \mid \kappa \to \kappa \\
\textit{types} & \tau & = & \alpha \mid \forall\alpha\!:\!\kappa.\tau \mid \tau_1 \multimap \tau_2 \mid\ !\tau \mid \lambda\alpha\!:\!\kappa.\tau \mid \tau_1\ \tau_2 \\
\textit{expressions} & e & = & x \mid \lambda\alpha\!:\!\kappa.e \mid e\ \tau \mid \lambda(\phi x)\!:\!\tau.e \mid e_1\ e_2 \mid\ !e \\
\textit{linearities} & \phi & = & \cdot \mid\ ! \\
\textit{type ctxts} & \Delta & = & \cdot \mid \Delta, \alpha : \kappa \\
\textit{value ctxts} & \Gamma & = & \cdot \mid \Gamma, \phi(x : \tau)
\end{array}
$$

Figure 6: Linear Fω Syntax

$$
\begin{array}{rcl}
\exists\alpha\!:\!\kappa.\tau & = & \forall\beta.(\forall\alpha\!:\!\kappa.\tau \multimap \beta) \multimap \beta \\
\tau_1 \otimes \tau_2 & = & \forall\alpha.(\tau_1 \multimap \tau_2 \multimap \alpha) \multimap \alpha \\
\tau_1 \times \tau_2 & = & !(\tau_1 \otimes \tau_2) \\
\tau_1 \to \tau_2 & = & !(\tau_1 \multimap \tau_2) \\
\tau_1 \rightrightarrows\!\!\!\circ\ \tau_2 & = & \tau_1 \multimap (\tau_2 \otimes (\tau_2 \multimap \tau_1)) \\
\tau_1 \rightrightarrows \tau_2 & = & \tau_1 \to (\tau_2 \otimes (\tau_2 \multimap \tau_1)) \\
(\circ) & = & \forall\alpha\!:\!\text{Type}.\alpha \multimap \alpha \\
() & = & !(\circ) \\
\text{true} & = & \exists\alpha\!:\!\text{Type}.\alpha
\end{array}
$$

Figure 7: Useful linear $F\omega$ type abbreviations

$$((\alpha \Leftrightarrow \emptyset) \times (\alpha_Z \Leftrightarrow \emptyset)) \vee (\alpha_Z \Leftrightarrow \text{true})$$

This invariant uses nonlinear pairs $C_1 \times C_2$, the nonlinear disjunctions (unions) $C_1 \vee C_2$, and nonlinear implications $C_1 \Rightarrow C_2$, where $C_1 \Leftrightarrow C_2$ is an abbreviation for $(C_1 \Rightarrow C_2) \times (C_2 \Rightarrow C_1)$. A nonlinear capability C_N can appear with a linear capability C_L inside a function type $((C_N \otimes C_L), \tau) \to 0$, but it's often convenient to split the nonlinear capability from the function type. For example, currying $((C_N \otimes C_L), \tau) \to 0$ yields the type $C_N \Rightarrow ((C_L, \tau) \to 0)$, where the type $C \Rightarrow \tau$ is analogous to a function type $\tau_1 \to \tau_2$. The translation of $\forall\alpha : \text{Cap}^+.\tau$ uses the type $C \Rightarrow \tau$ to ensure that the invariant above is satisfied for every instantiation of α and α_Z:

$$\mathcal{T}(\forall\alpha\!:\!\text{Cap}^+.\tau) = \forall\alpha\!:\!\text{Cap}.\forall\alpha_Z\!:\!\text{Cap}.(((\alpha \Leftrightarrow \emptyset) \times (\alpha_Z \Leftrightarrow \emptyset)) \vee (\alpha_Z \Leftrightarrow \text{true})) \Rightarrow \mathcal{T}(\tau)$$

Further details of the translation appear in appendix C.

By using $\mathcal{Z}(A)$ to track precisely when A is equivalent to $\emptyset$, the revised translation ensures that the program only halts when the current capability is equivalent to $\emptyset$.

7 From CC2 to linear $F\omega$

Section 6 demonstrated that LC can express CC0's aliasing without relying on CC0's capability syntax and capability rules. Nevertheless, LC shares much of its syntax with CC0, and this raises a theoretical question: which features in LC and CC0 are essential for expressing CC0's aliasing? Does a target language powerful enough to encode CC0 in well-typed way require...

1. ...a distinction between handles and capabilities?
2. ...the linear choice operator (e.g. in the rule for the "use v" expression)?
3. ...separate proof and programming languages?
4. ...a large set of features?

This section demonstrates that (1), (2), and (3) are not essential (though they are useful in practice), and that the target language can be very small. Figure 6 shows linear $F\omega$ (the higher-order polymorphic lambda calculus $F\omega$ extended with linear types). Linear $F\omega$ contains only 2 basic types for values: $\tau_1 \multimap \tau_2$ for linear functions and $\forall\alpha:\kappa.\tau$ for linear polymorphic abstractions. The "of course" operator ! turns linear types into nonlinear types; $!(\tau_1 \multimap \tau_2)$ is the type for nonlinear functions (abbreviated here as $\tau_1 \to \tau_2$), and $!\forall\alpha : \kappa.\tau$ is the type for nonlinear polymorphic abstractions. The polymorphic lambda calculus has the remarkable ability to encode many other types just using function types and polymorphic types; figure 7 shows some standard encodings of standard types (as well as a couple nonstandard types, $\tau_1 \rightleftarrows \tau_2$ and $\tau_1 \rightrightarrows \tau_2$, explained below).

Linear $F\omega$ lacks any built-in types for linear resources, such as files, or expressions for manipulating linear resources. Instead, the translation assumes that the type environment contains an abstract type constructor χ for linear resources, and the value environment contains variables new, $free$, and use that act on linear resources:

$$new : () \to \exists\rho{:}\text{Type}.(\chi\ \rho)$$
$$free : !\forall\rho{:}\text{Type}.(\chi\ \rho) \to ()$$
$$use : !\forall\rho{:}\text{Type}.(\chi\ \rho) \to (\chi\ \rho)$$

Unlike CC0's operations, which manipulate linear capabilities and nonlinear handles, the operations shown above do not distinguish between capabilities and handles (the linear type $\chi\ \rho$ contains both the run-time information about a resource and the permission to use the resource). Section 7.1 describes a particular instantiation of new, $free$, and use for alias types; interestingly, implementing these operations for alias types, and even adding operations for alias type mutation, requires no extensions to linear $F\omega$.

The translation from CC2's unique capabilities U to linear $F\omega$'s types follows the strategy from section 6, using linear pairs to implement CC2's join operator:

$$\mathcal{U}(\alpha) = \alpha$$
$$\mathcal{U}(\emptyset) = ()$$
$$\mathcal{U}(\{\rho^{\varphi}\}) = \chi\ \rho$$
$$\mathcal{U}(U_1 \oplus U_2) = \mathcal{U}(U_1) \otimes \mathcal{U}(U_2)$$

The translation of alias capabilities A to linear $F\omega$ follows the *hello* example from section 2.1.1:

$$hello : \forall\rho_1.\forall\rho_2.\forall\alpha.\forall\beta_1.\forall\beta_2.\alpha \otimes (\alpha \to \beta_1 \otimes \rho_1\,\text{cap}) \otimes (\beta_1 \otimes \rho_1\,\text{cap} \to \alpha) \otimes \rho_1\,\text{handle}$$
$$\otimes(\alpha \to \beta_2 \otimes \rho_2\,\text{cap}) \otimes (\beta_2 \otimes \rho_2\,\text{cap} \to \alpha) \otimes \rho_2\,\text{handle} \to \alpha$$

In *hello*'s type, the type variable α serves as a pool of capabilities from which the particular capabilities ρ_1 cap and ρ_2 cap can be extracted. An equivalent way to write *hello*'s type is:

$$hello : \forall\rho_1.\forall\rho_2.\forall\alpha.\alpha \otimes (\alpha \to \exists\beta_1.\beta_1 \otimes \rho_1\,\text{cap} \otimes (\beta_1 \otimes \rho_1\,\text{cap} \to \alpha)) \otimes \rho_1\,\text{handle}$$
$$\otimes(\alpha \to \exists\beta_2.\beta_2 \otimes \rho_2\,\text{cap} \otimes (\beta_2 \otimes \rho_2\,\text{cap} \to \alpha)) \otimes \rho_2\,\text{handle} \to \alpha$$

In this form, the β_1 and β_2 can hide inside linear functions, yielding a simpler type:

$$hello : \forall\rho_1.\forall\rho_2.\forall\alpha.\alpha \otimes (\alpha \to \rho_1\,\text{cap} \otimes (\rho_1\,\text{cap} \multimap \alpha)) \otimes \rho_1\,\text{handle}$$
$$\otimes(\alpha \to \rho_2\,\text{cap} \otimes (\rho_2\,\text{cap} \multimap \alpha)) \otimes \rho_2\,\text{handle} \to \alpha$$

For convenience, the abbreviation $\tau_1 \rightrightarrows \tau_2 = \tau_1 \rightarrow (\tau_2 \otimes (\tau_2 \multimap \tau_1))$ expresses the idea that a small linear resource τ_2 can be temporarily extracted from a larger linear resource τ_1 (note that even though τ_1 and τ_2 may be linear, $\tau_1 \rightrightarrows \tau_2$ is nonlinear). With this abbreviation, *hello*'s type becomes:

$$\begin{aligned} \mathit{hello} : \forall\rho_1.\forall\rho_2.\forall\alpha.\alpha \otimes (\alpha \rightrightarrows \rho_1\, \text{cap}) \otimes \rho_1\, \text{handle} \\ \otimes(\alpha \rightrightarrows \rho_2\, \text{cap}) \otimes \rho_2\, \text{handle} \rightarrow \alpha \end{aligned}$$

The type for *hello* suggests a translation for CC2's join operator:

$$\mathcal{A}(\{\rho_1^+\} \oplus \{\rho_2^+\}) = \exists\alpha{:}\text{Type}.\alpha \otimes (\alpha \rightrightarrows \chi\ \rho_1) \otimes (\alpha \rightrightarrows \chi\ \rho_2)$$

At first glance, this type satisfies the most crucial test for an encoding of CC0: unique capabilities can be coerced to alias capabilities. For example, to encode $\{\rho_1^1\} \oplus \{\rho_2^1\} \leq \{\rho_1^+\} \oplus \{\rho_2^+\}$, there is a function of type:

$$(\chi\ \rho_1) \otimes (\chi\ \rho_2) \rightarrow (\exists\alpha{:}\text{Type}.\alpha \otimes (\alpha \rightrightarrows \chi\ \rho_1) \otimes (\alpha \rightrightarrows \chi\ \rho_2))$$

Similarly, to encode $\{\rho^1\} \leq \{\rho^+\} \oplus \{\rho^+\}$, there is a function of type:

$$(\chi\ \rho) \rightarrow (\exists\alpha{:}\text{Type}.\alpha \otimes (\alpha \rightrightarrows \chi\ \rho) \otimes (\alpha \rightrightarrows \chi\ \rho))$$

In a proof language, this would be sufficient to express CC0's subcapability relation. A programming language, though, has a harder task: it's not enough to say that $(\chi\ \rho_1)\otimes(\chi\ \rho_2)$ *can be* coerced to type $\mathcal{A}(\{\rho_1^+\} \oplus \{\rho_2^+\})$; a running program must actually *perform* the coercion if it wants to use a run-time value of type $\mathcal{A}(\{\rho_1^+\}\oplus\{\rho_2^+\})$. More onerously, it must be able to get back the original value of type $(\chi\ \rho_1) \otimes (\chi\ \rho_2)$ after using $\mathcal{A}(\{\rho_1^+\} \oplus \{\rho_2^+\})$ — the whole point of CC0's bounded quantification is to view resources as temporarily aliased, and then later restore the resources' uniqueness. Unfortunately, the reverse direction of the functions shown above does not hold; there is no value of this type:

$$(\chi\ \rho_1) \otimes (\chi\ \rho_2) \leftrightarrow (\exists\alpha{:}\text{Type}.\alpha \otimes (\alpha \rightrightarrows \chi\ \rho_1) \otimes (\alpha \rightrightarrows \chi\ \rho_2))$$

(Here, $\tau_1 \leftrightarrow \tau_2$ is an abbreviation $(\tau_1 \rightarrow \tau_2) \times (\tau_2 \rightarrow \tau_1)$.) On the other hand, there is a value of the type shown below, where the scope of α is wide enough to ensure that both the forward and reverse functions agree on the same α:

$$\exists\alpha{:}\text{Type}.(\chi\ \rho_1) \otimes (\chi\ \rho_2) \leftrightarrow (\alpha \otimes (\alpha \rightrightarrows \chi\ \rho_1) \otimes (\alpha \rightrightarrows \chi\ \rho_2))$$

This small change in scope puts a large administrative burden on the encoding, because the declaration of α now sits outside the definition of $\mathcal{A}(\{\rho_1^+\}\oplus\{\rho_2^+\})$. Therefore, $\mathcal{A}(\{\rho_1^+\}\oplus\{\rho_2^+\})$ must be parameterized over α. In other words, $\mathcal{A}(\{\rho_1^+\} \oplus \{\rho_2^+\})$ is no longer a type of kind Type, but is instead a type constructor of kind Type $\rightarrow$ Type:

$$\mathcal{A}(\{\rho_1^+\} \oplus \{\rho_2^+\}) = \lambda\gamma{:}\text{Type}.\gamma \otimes (\gamma \rightrightarrows \chi\ \rho_1) \otimes (\gamma \rightrightarrows \chi\ \rho_2)$$

The translation turns CC2's kind Cap^+ into kind Type $\rightarrow$ Type, while all other CC2 kinds become Type:

$$\begin{aligned} &\mathcal{K}(\text{Cap}^1) = \text{Type} \\ &\mathcal{K}(\text{Cap}^+) = \text{Type} \rightarrow \text{Type} \\ &\mathcal{K}(\text{Type}) = \text{Type} \\ &\mathcal{K}(\text{Res}) = \text{Type} \end{aligned}$$

Once γ is a parameter to $\mathcal{A}(\{\rho_1^+\} \oplus \{\rho_2^+\})$ and the "$\exists\gamma{:}\text{Type}.$" sits outside $\mathcal{A}(\{\rho_1^+\} \oplus \{\rho_2^+\})$, it's convenient to pull the "$\gamma\otimes$" outside as well, so that what's left inside $\mathcal{A}(\{\rho_1^+\} \oplus \{\rho_2^+\})$ is purely nonlinear (and thus easier to manipulate):

$$\mathcal{A}(\{\rho_1^+\} \oplus \{\rho_2^+\}) = \lambda\gamma\!:\!\text{Type}.(\gamma \rightrightarrows \chi\ \rho_1) \times (\gamma \rightrightarrows \chi\ \rho_2)$$

The following definitions extend the $\mathcal{A}(\{\rho_1^+\} \oplus \{\rho_2^+\})$ example to general $\mathcal{A}(A)$:

$$\begin{array}{l}
\mathcal{A}(\alpha) = \lambda\gamma\!:\!\text{Type}.\,!(\alpha\ \gamma) \\
\mathcal{A}(\emptyset) = \lambda\gamma\!:\!\text{Type}.() \\
\mathcal{A}(\{\rho^\varphi\}) = \lambda\gamma\!:\!\text{Type}.(\gamma \rightrightarrows \chi\ \rho) \\
\mathcal{A}(A_1 \oplus A_2) = \lambda\gamma\!:\!\text{Type}.(\mathcal{A}(A_1)\ \gamma) \times (\mathcal{A}(A_2)\ \gamma)
\end{array}$$

Two small improvements are worth making to these definitions. First, it is convenient to pull the nonlinear "of course" operator (" !") outside the definition of $\mathcal{A}(A)$, so that the definition of $\mathcal{A}(\alpha)$ becomes $\mathcal{A}(\alpha) = \lambda\gamma\!:\!\text{Type}.(\alpha\ \gamma)$, or, more simply, just $\mathcal{A}(\alpha) = \alpha$:

$$\begin{array}{l}
\mathcal{A}(\alpha) = \alpha \\
\mathcal{A}(\emptyset) = \lambda\gamma\!:\!\text{Type}.(\!\circ\!) \\
\mathcal{A}(\{\rho^\varphi\}) = \lambda\gamma\!:\!\text{Type}.(\gamma \rightrightarrows\!\!\!\circ\ \chi\ \rho) \\
\mathcal{A}(A_1 \oplus A_2) = \lambda\gamma\!:\!\text{Type}.\,!(\mathcal{A}(A_1)\ \gamma) \otimes !(\mathcal{A}(A_2)\ \gamma)
\end{array}$$

Second, forcing all the $\mathcal{A}(A)$ to use the same γ is overly restrictive, and makes it difficult to translate CC2's equality and subcapability rules. The following definitions give $\mathcal{A}(A_1 \oplus A_2)$ the flexibility to refine γ so that A_1 and A_2 can use smaller pieces of γ. This flexibility makes it easier to glue $\mathcal{A}(A_1)$ and $\mathcal{A}(A_2)$ together to form $\mathcal{A}(A_1 \oplus A_2)$.

$$\begin{array}{l}
\mathcal{A}(A) = \lambda\gamma\!:\!\text{Type}.\exists\delta\!:\!\text{Type}.(\gamma \rightrightarrows \delta) \times !(\mathcal{A}[A]\ \delta) \\
\mathcal{A}[\alpha] = \alpha \\
\mathcal{A}[\emptyset] = \lambda\gamma\!:\!\text{Type}.(\!\circ\!) \\
\mathcal{A}[\{\rho^\varphi\}] = \lambda\gamma\!:\!\text{Type}.\gamma \rightrightarrows\!\!\!\circ\ \chi\ \rho \\
\mathcal{A}[A_1 \oplus A_2] = \lambda\gamma\!:\!\text{Type}.\,!(\mathcal{A}(A_1)\ \gamma) \otimes !(\mathcal{A}(A_2)\ \gamma)
\end{array}$$

Given the definitions of $\mathcal{U}(U)$ and $\mathcal{A}(A)$, most of the type translation is straightforward:

$$\begin{array}{l}
\mathcal{T}(\alpha) = \alpha \\
\mathcal{T}(\tau_1 \times \tau_2) = !\mathcal{T}(\tau_1) \otimes !\mathcal{T}(\tau_2) \\
\mathcal{T}((U \boxplus A, \tau) \to 0) = \forall\gamma\!:\!\text{Type}.\gamma \to \mathcal{U}(U) \multimap !(\mathcal{A}(A)\ \gamma) \multimap !\mathcal{T}(\tau) \multimap \text{true} \\
\mathcal{T}(\rho\,\text{handle}) = (\!\circ\!) \\
\mathcal{T}(\forall\alpha\!:\!\kappa.\tau) = \forall\alpha\!:\!\mathcal{K}(\kappa).\,!\mathcal{T}(\tau)
\end{array}$$

Since a resource's run-time information resides in a value of type $\chi\ \rho$, there's no need for a separate handle value for the resource, so the translation of type $\rho\,$handle is empty. The translation $\mathcal{T}((U \boxplus A, \tau) \to 0)$ defines a curried function that takes arguments of type $\mathcal{U}(U)$, $!(\mathcal{A}(A)\ \gamma)$, and $!\mathcal{T}(\tau)$, plus the pool γ from which $!(\mathcal{A}(A)\ \gamma)$ extracts capabilities. The function's return type "true" allows the function to discard the pool γ when the program halts; this does not capture the complete collection property, but it's likely that the translation could be revised to express complete collection using the techniques from section 6.2.

To implement bounded quantification, the translation uses the $(\forall\alpha \le \tau'.\tau) = \forall\alpha.(\alpha \to \tau') \to \tau$ encoding from section 6:

$$\begin{array}{l}
\mathcal{T}(\forall\alpha\!:\!\text{Cap}^+\ \text{where}\ C_1 \le A_1, \ldots, C_n \le A_n.\tau) = \\
\qquad \forall\alpha\!:\!\text{Type} \to \text{Type}.\,!\mathcal{S}(C_1 \le A_1) \to \ldots \to !\mathcal{S}(C_n \le A_n) \to !\mathcal{T}(\tau)
\end{array}$$

This definition relies on an encoding $!\mathcal{S}(C_1 \le A_1)$ of CC2's subcapability relation, as described below.

Linear $F\omega$ lacks the rich set of type equivalence, capability equivalence, and subcapability judgments found in CC0. Following [1], the translation into linear $F\omega$ encodes these judgments as expressions. For example, if two CC2 types τ_1 and τ_2 are equivalent, then the translation produces an expression e

$$\Delta \vdash \tau_1 = \tau_2 : \text{Type} \rightsquigarrow e$$

such e has type $!\mathcal{T}(\tau_1) \leftrightarrow !\mathcal{T}(\tau_2)$. (See appendix C for the complete definition of $\Delta \vdash \tau_1 = \tau_2 : \text{Type} \rightsquigarrow e$.) Similarly, the translation encodes capability equivalence $U_1 = U_2$ and $A_1 = A_2$ as expressions of type $\mathcal{U}(U_1) \leftrightarrow \mathcal{U}(U_2)$ and $!\forall\gamma{:}\text{Type}.!(\mathcal{A}(A_1)\ \gamma) \leftrightarrow !(\mathcal{A}(A_2)\ \gamma)$.

Encoding subcapabilities is slightly more interesting. Suppose that a function f_1 of type $((U \oplus U_1) \boxplus A_1, \tau) \to 0$ wants to call a function f_2 of type $(U \boxplus A_2, \tau) \to 0$, where $U_1 \boxplus A_1 \leq A_2$. The translations of f_1's type and f_2's type are

$$\mathcal{T}(((U{\oplus}U_1){\boxplus}A_1, \tau) \to 0) = \forall\gamma{:}\text{Type}.\gamma \to \mathcal{U}(U){\otimes}\mathcal{U}(U_1) \multimap !(\mathcal{A}(A_1)\ \gamma) \multimap !\mathcal{T}(\tau) \multimap \text{true}$$

$$\mathcal{T}((U \boxplus A_2, \tau) \to 0) = \forall\gamma'{:}\text{Type}.\gamma' \to \mathcal{U}(U) \multimap !(\mathcal{A}(A_2)\ \gamma') \multimap !\mathcal{T}(\tau) \multimap \text{true}$$

While f_2 accepts only two linear arguments, γ' and $\mathcal{U}(U)$, f_1 holds three linear values, γ, $\mathcal{U}(U)$, and $\mathcal{U}(U_1)$. Clearly, f_1 should pass its own $\mathcal{U}(U)$ value as the $\mathcal{U}(U)$ argument to f_2. This leaves f_1's other two values, γ and $\mathcal{U}(U_1)$, to instantiate f_2's γ' argument, so f_1 should choose $\gamma' = \gamma \otimes \mathcal{U}(U_1)$. Now f_1 needs to instantiate f_2's nonlinear argument $!(\mathcal{A}(A_2)\ (\gamma \otimes \mathcal{U}(U_1)))$, but f_1 only holds a value of type $!(\mathcal{A}(A_1)\ \gamma)$. This is where the translation of $U_1 \boxplus A_1 \leq A_2$ comes in — to allow f_1 to call f_2, encode $U_1 \boxplus A_1 \leq A_2$ as an expression of type:

$$\mathcal{S}(U_1 \boxplus A_1 \leq A_2) = \forall\gamma{:}\text{Type}.!(\mathcal{A}(A_1)\ \gamma) \to !(\mathcal{A}(A_2)\ (\gamma \otimes \mathcal{U}(U_1)))$$

As an example, consider the CC2 typing rule for the "use v" expression:

$$\frac{\Delta;\Gamma \vdash v : \alpha\,\text{handle} \qquad \Delta \vdash U = U_B \oplus U' : \text{Cap}^1 \qquad \Delta \vdash U_B \boxplus A \leq A' \oplus \{\alpha^+\}}{\Delta;\Gamma;U \boxplus A \vdash \text{use}\,v \Longrightarrow \Delta;\Gamma;U \boxplus A}$$

The encoded type for the $U_B \boxplus A \leq A' \oplus \{\alpha^+\}$ that appears in the typing rule is:

$$\mathcal{S}(U_B \boxplus A \leq A' \oplus \{\alpha^+\}) = \forall\gamma{:}\text{Type}.!(\mathcal{A}(A)\ \gamma) \to !(\mathcal{A}(A' \oplus \{\alpha^+\})\ (\gamma \otimes \mathcal{U}(U_B)))$$

The definition of $\mathcal{A}(A' \oplus \{\alpha^+\})\ (\gamma \otimes \mathcal{U}(U_B))$ is:

$$\begin{aligned}
&\mathcal{A}(A' \oplus \{\alpha^+\})\ (\gamma \otimes \mathcal{U}(U_B)) = \exists\delta{:}\text{Type}.((\gamma \otimes \mathcal{U}(U_B)) \rightrightarrows \delta) \times !(\mathcal{A}[A' \oplus \{\alpha^+\}]\ \delta)\\
&\mathcal{A}[A' \oplus \{\alpha^+\}]\ \delta = !(\mathcal{A}(A')\ \delta) \otimes !(\mathcal{A}(\{\alpha^+\})\ \delta)\\
&\mathcal{A}(\{\alpha^+\})\ \delta = \lambda\epsilon{:}\text{Type}.\exists\delta{:}\text{Type}.(\delta \rightrightarrows \epsilon) \times !(\mathcal{A}[\{\alpha^+\}]\ \epsilon)\\
&\mathcal{A}[\{\alpha^+\}]\ \epsilon = \epsilon \rightrightarrows \chi\ \alpha
\end{aligned}$$

Using a value of type $\mathcal{A}(A' \oplus \{\alpha^+\})\ (\gamma \otimes \mathcal{U}(U_B))$, the translation of "use v" retrieves values of type $(\gamma \otimes \mathcal{U}(U_B)) \rightrightarrows \delta$, $\delta \rightrightarrows \epsilon$, and $\epsilon \rightrightarrows \chi\ \alpha$. Because the extraction operator $\rightrightarrows$ is transitive, the translation combines these into a single value of type $(\gamma \otimes \mathcal{U}(U_B)) \rightrightarrows \chi\ \alpha$, from which it temporarily extracts the resource $\chi\ \alpha$ so that it can call the *use* function:

$$use : !\forall\rho{:}\text{Type}.(\chi\ \rho) \to (\chi\ \rho)$$

7.1 Alias types in pure linear $F\omega$

So far, this paper has treated resources as abstract, assuming only some generic operations "new", "free", and "use" on resources. This section replaces abstract resources with a particular concrete resource, mutable linear heap objects, in order to implement the malloc, load, store, and free operations of alias types [24]. For simplicity, each heap object will be a linear pair, rather than an arbitrary-size linear tuple. To start with, consider the *new*, *free*, and *use* functions targeted by section 7's translation:

$$
\begin{aligned}
&\mathit{new} : () \rightarrow \exists\rho{:}\mathrm{Type}.(\chi\ \rho)\\
&\mathit{free} : !\forall\rho{:}\mathrm{Type}.(\chi\ \rho) \rightarrow ()\\
&\mathit{use} : !\forall\rho{:}\mathrm{Type}.(\chi\ \rho) \rightarrow (\chi\ \rho)
\end{aligned}
$$

Define the abstract type χ to be a linear pair:

$$\chi = \lambda\rho{:}\mathrm{Type}.\tau_1 \otimes \tau_2$$

Suppose that $\tau_1 = ()$ and $\tau_2 = ()$. Then the following definitions of *new*, *free*, and *use* are correctly typed, though rather boringly implemented:

$$
\begin{aligned}
&\mathit{new} = !\lambda\, !x{:}().\mathrm{pack}[(), \langle (), () \rangle]\ \mathrm{as}\ \exists\rho{:}\mathrm{Type}.() \otimes ()\\
&\mathit{free} = !\lambda\rho{:}\mathrm{Type}.\, !\lambda\langle\, !x_1,\ !x_2\rangle : () \otimes ().()\\
&\mathit{use} = !\lambda\rho{:}\mathrm{Type}.\, !\lambda x{:}() \otimes ().x
\end{aligned}
$$

(These definitions rely on encodings of "pack", pair operations, and pattern matching, as defined in appendix B.)

Unlike the capability calculus, whose capabilities only track the existence or non-existence of a resource ρ, alias types associate a state with each resource, so that capabilities have the form $\{\rho \mapsto \mathit{state}^{\varphi}\}$, rather than just $\{\rho^{\varphi}\}$. For the example of linear heap pairs, the state in each capability is the type of the two fields of the pair: $\{\rho \mapsto \tau_1 \otimes \tau_2^{\varphi}\}$. To track this state, the resource type and resource operations must be parameterized over all possible τ_1 and τ_2:

$$
\begin{aligned}
&\chi = \lambda\rho{:}\mathrm{Type}.\lambda\beta_1{:}\mathrm{Type}.\lambda\beta_2{:}\mathrm{Type}.\, !\beta_1 \otimes !\beta_2\\
&\mathit{new} : () \rightarrow \exists\rho{:}\mathrm{Type}.(\chi\ \rho\ (\circ)\ (\circ))\\
&\mathit{free} : !\forall\rho{:}\mathrm{Type}.\, !\forall\beta_1{:}\mathrm{Type}.\, !\forall\beta_2{:}\mathrm{Type}.(\chi\ \rho\ \beta_1\ \beta_2) \rightarrow ()\\
&\mathit{use} : !\forall\rho{:}\mathrm{Type}.\, !\forall\beta_1{:}\mathrm{Type}.\, !\forall\beta_2{:}\mathrm{Type}.(\chi\ \rho\ \beta_1\ \beta_2) \rightarrow (\chi\ \rho\ \beta_1\ \beta_2)
\end{aligned}
$$

Furthermore, it's useful to have *use* actually return a value from a heap object, rather than simply touching the object. Replace *use* with two functions load_1 and load_2, which read the first and second fields of a heap pair:

$$\mathit{load}_k : !\forall\rho{:}\mathrm{Type}.\, !\forall\beta_1{:}\mathrm{Type}.\, !\forall\beta_2{:}\mathrm{Type}.(\chi\ \rho\ \beta_1\ \beta_2) \rightarrow (\chi\ \rho\ \beta_1\ \beta_2) \otimes !\beta_k$$

It's straightforward to update the CC0-to-linear $F\omega$ translations to target the new versions of χ, *new*, *free*, and *use* (*load*). Furthermore, it's also easy to support another operation, *store*:

$$
\begin{aligned}
&\mathit{store}_1 : !\forall\rho{:}\mathrm{Type}.\, !\forall\beta_1{:}\mathrm{Type}.\, !\forall\beta_2{:}\mathrm{Type}.\, !\forall\beta'{:}\mathrm{Type}.(\chi\ \rho\ \beta_1\ \beta_2) \otimes !\beta \rightarrow (\chi\ \rho\ \beta'\ \beta_2)\\
&\mathit{store}_2 : !\forall\rho{:}\mathrm{Type}.\, !\forall\beta_1{:}\mathrm{Type}.\, !\forall\beta_2{:}\mathrm{Type}.\, !\forall\beta'{:}\mathrm{Type}.(\chi\ \rho\ \beta_1\ \beta_2) \otimes !\beta \rightarrow (\chi\ \rho\ \beta_1\ \beta')
\end{aligned}
$$

where the CC0 typing rule for store is a slight variation on the rules for new and free:

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash v : \rho\,\mathrm{handle} \qquad \Delta;\Gamma \vdash v' : \tau' \\ \Delta \vdash C = C' \oplus \{\rho \mapsto \tau_1 \otimes \tau_2^1\} : \mathrm{Cap}\end{array}}{\Delta;\Gamma;C \vdash \mathrm{store}\, v[k] \leftarrow v' \Longrightarrow \Delta;\Gamma;C' \oplus \{\rho \mapsto \tau_1' \otimes \tau_2'^1\}}\,(\tau_k' = \tau',\ \tau_j' = \tau_j\,, j \neq k)$$

(Because the rule for store does not involve subcapabilities or alias capabilities, "store" is no more difficult to translate than "new" and "free", and is much easier to translate than "use".) Given a revised translation targeting χ, *new*, *free*, *load*, and *store*, the following definitions are correctly typed:

$$
\begin{aligned}
&\mathit{new} = !\lambda\, !x{:}().\mathrm{pack}[(), \langle (), () \rangle]\ \mathrm{as}\ \exists\rho{:}\mathrm{Type}.() \otimes ()\\
&\mathit{free} = !\lambda\rho{:}\mathrm{Type}.\, !\lambda\beta_1{:}\mathrm{Type}.\, !\lambda\beta_2{:}\mathrm{Type}.\, !\lambda\langle\, !x_1,\ !x_2\rangle : !\beta_1 \otimes !\beta_2.()\\
&\mathit{load}_k = !\lambda\rho{:}\mathrm{Type}.\, !\lambda\beta_1{:}\mathrm{Type}.\, !\lambda\beta_2{:}\mathrm{Type}.\, !\lambda\langle\, !x_1,\ !x_2\rangle : !\beta_1 \otimes !\beta_2.\langle\langle x_1, x_2\rangle, x_k\rangle\\
&\mathit{store}_k = !\lambda\rho{:}\mathrm{Type}.\, !\lambda\beta_1{:}\mathrm{Type}.\, !\lambda\beta_2{:}\mathrm{Type}.\, !\lambda\beta'{:}\mathrm{Type}.\, !\lambda\langle\langle\, !x_1,\ !x_2\rangle, !x'\rangle :\\
&(\, !\beta_1 \otimes !\beta_2) \otimes \beta'.\langle x_1', x_2'\rangle\\
&\qquad \mathit{where}\ x_k' = x'\ \mathit{and}\ x_j' = x_j\ \mathit{for}\ j \neq k
\end{aligned}
$$

8 Related work

This paper demonstrates that linear type systems can encode aliasing (multiple references to a linear resource) even if the linear type system forces the program to have only a *single reference* to each linear resource at each step of the program's execution (as for example, LC and linear $F\omega$ do). Other papers [4][26] have pointed out that some implementations do not enforce the "single reference" property for some linear expressions, which leads to another form of aliasing. Suppose that the expression "let $!x = !\langle 2,3\rangle$ in $!\langle x,x\rangle$" steps to a new expression "$!\langle\langle 2,3\rangle,\langle 2,3\rangle\rangle$". The new expression holds two linear values, both equal to $\langle 2,3\rangle$, and both have a single reference to them. An efficient implementation of this expression, though, might only create a single $\langle 2,3\rangle$ value, rather than making two copies of the value. In this case, there would be two references to the shared $\langle 2,3\rangle$ value. This sharing does not apply to linear resources in general, though — if z has linear type $File$, then "let $!x = !z$ in $!\langle x,x\rangle$" is ill-typed, because the linear variable z is not available when type-checking the nonlinear expression $!z$. Therefore, it's not clear that this form of aliasing is useful as a programming technique for general linear resources, even though it may have an impact on compiler design and run-time system design, depending on the details of the linear type system.

Wadler [27] described a "let!" expression that allowed temporary aliasing of a linear resource. This expression's typing rule relied on an unusual constraint on the type of its bound variable, so it would be interesting to see if there is an encoding of this expression using conventional linear types.

Fluet and Morrisett [11] used monads to encode a variant of Tofte and Talpin's region calculus [25]. In particular, they represented Tofte and Talpin's "letregion" construct using a monadic operation "letRGN" that allows an expression to build arbitrary state transformations on a region s, which is allocated before the transformations take place and deallocated after the transformations complete. The state transformers for s must be polymorphic over all s, which is not in scope in the type of the transformation's final result. This guarantees that the transformers cannot leak s to the outside world, so that no dangling references to s are possible. If state transformers were only allowed to access one region at a time, then this approach would just be monadic way of expressing s's linearity. However, the state transformers for s also have access to s's enclosing regions, which may be aliased. Effectively, the transformers see s with kind Cap^1 and s's enclosing regions $r_1,\ldots,r_n$ with kind Cap^+. This seems similar to section 2.1's approach of treating aliased files (corresponding to the aliased $r_1,\ldots,r_n$) as nonlinear references into a linear pool α (corresponding to the linear s). It also seems related to Fähndrich and Deline's idea of a linear object s "adopting" other objects $r_1,\ldots,r_n$ [10]. This connection suggests that some of letRGN's limitations (such as a LIFO ordering on region allocation/deallocation) are not fundamental.

Walker and Watkins [31] described how to use linear types to manipulate regions (as an alternative to using the capability calculus for regions). On the positive side, first-class linear types allowed many idioms, such as heterogeneous data structures, that were not easily expressible in the capability calculus. On the negative side, unlike the capability calculus, their language was not able to encode Tofte and Talpin's region calculus (although it could, by using a form of Wadler's "let!" expression, encode a simplified form of Tofte and Talpin's "letregion" construct).

Crary and Vanderwaart [6] describe how to represent the capability calculus in their language LTT. Like the CC0-to-LC encoding in this paper, their representation relies heavily on linear types. Unlike the CC0-to-LC encoding, though, they introduce CC0's equality and subcapability rules as axioms rather than deriving CC0's rules from linear type rules, so that the soundness of their system requires a proof of CC0's soundness.

Morrisett, Ahmed, and Fluet [18] provide denotational semantics for L^3, a subset of alias types based on linear types. In addition to proving soundness, the semantics show that all well-typed L^3 programs terminate. The CC0-to-linear $F\omega$ translation could also serve as denotational semantics for alias types (albeit a non-set-theoretic semantics). In particular,

like the L^3 semantics, the CC0-to-linear $F\omega$ translation demonstrates that well-typed programs terminate. The CC0-to-linear $F\omega$ translation is, admittedly, far more complicated than the L^3 semantics. Then again, L^3 lacks CC0's duplicable capabilities, CC0's capability equality rules, and CC0's subcapability rules; almost all of the work in the CC0-to-linear $F\omega$ translation is devoted to handling these CC0 features.

Cheney and Morrisett [3] compile a nonlinear source language into a linear target language. They eliminate aliasing by copying aliased data structures. Their copying technique does not apply to general linear resources, though — a compiler can't generate copies of a hardware device, for example, nor can it duplicate mutable data structures without changing the semantics of mutation. For these reasons, the CC0-to-LC and CC0-to-linear $F\omega$ translations do not try to copy aliased linear resources.

Jia and Walker describe ILC- [15], a decidable program logic with support for linearity (this was inspired partly by separation logic [14][22]). Hopefully, some of the techniques in the CC0-to-LC translation are applicable to ILC-, so that programs based on ILC- can employ CC0's style of aliasing.

Hawblitzel, Huang, and Wittie [13] use the technique from sections 2.1 and 2.1.1 to encode aliased pointers inside a region [25]. Unfortunately, their mechanism for allocation in regions prevented them from using an elegant proof language, such as LC's proof language. They did not deal with CC0's capability equality and subcapability rules.

9 Conclusions

In a narrow, literal sense, linear type systems disallow aliasing of linear resources. The semantics of the language can state and prove this literal prohibition precisely. In practice, though, linear type systems can faithfully emulate common aliasing idioms. Therefore, it is not necessary to add new type system features to support these idioms, as CC0 does; programs can simply tap the aliasing already inherent in linear type systems.

Of course, the encodings presented in this paper are not simple. This complexity, though, is mostly a reflection of how elaborate CC0's rules are, and it's not clear that most programs need the full power of CC0's type system. For example, the *hello* function in section 2 was quite simple. As another example, alias types [23] omitted many of CC0's features, such as a stripping operator. Encoding aliasing with linear types need not be complicated, but a linear type system is powerful enough to let the encoding grow in complexity as needed.

In his pioneering 1990 paper [27], Wadler announced that "Linear types can change the world!". On the other hand, in a later tutorial [28], he used linear logic to express the dour wisdom that you can't both have your cake and eat it ($Cake \multimap Full, Cake \not\vdash Cake \otimes Full$). This paper concludes on a more upbeat sentiment: with linear types, you can have your world and alias it, too.

Acknowledgments

The author would like to thank David Walker for his inspiration and suggestions for developing the translation into LC.

References

[1] Martin Abadi and Marcelo P. Fiore. Syntactic considerations on recursive types. In *LICS '96: Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, page 242, Washington, DC, USA, 1996. IEEE Computer Society.

[2] Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. Inheritance as implicit coercion. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects*

of Object-Oriented Programming: Types, Semantics, and Language Design, pages 197–245. The MIT Press, Cambridge, MA, 1994.

[3] James Cheney and Greg Morrisett. A linearly typed assembly language. Technical report, Department of Computer Science, Cornell University.

[4] Jawahar Chirimar, Carl A. Gunter, and Jon G. Riecke. Reference counting as a computational interpretation of linear logic. *Journal of Functional Programming*, 6(2):195–244, 1996.

[5] Karl Crary. Typed compilation of inclusive subtyping. In *2000 ACM SIGPLAN International Conference on Functional Programming*, pages 68–81, 2000.

[6] Karl Crary and Joseph C. Vanderwaart. An expressive, scalable type theory for certified code. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 191–205. ACM Press, 2002.

[7] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 262–275. ACM Press, 1999.

[8] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. Technical Report TR2000-1780, Cornell University, 2000.

[9] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 59–69, 2001.

[10] M. Fähndrich and R. DeLine. Adoption and focus: Practical linear types for imperative programming. In *Proceedings of the SIGPLAN'02 Conference on Programming Language Design and Implementation*, June 2002.

[11] Matthew Fluet and Greg Morrisett. Monadic regions. In *ICFP '04: Proceedings of the ninth ACM SIGPLAN international conference on Functional programming*, pages 103–114, New York, NY, USA, 2004. ACM Press.

[12] Chris Hawblitzel. http://research.microsoft.com/~chrishaw/linearaliasing/proofs/, 2005.

[13] Chris Hawblitzel, Edward Wei, Heng Huang, Eric Krupski, and Lea Wittie. Low-level linear memory management. In *Workshop on Semantics, Program Analysis, and Computing Environments For Memory Management*, 2004.

[14] Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *Symposium on Principles of Programming Languages*, pages 14–26, 2001.

[15] Limin Jia and David Walker. A refined proof theory for reasoning about separation. In Prakash Panangaden, editor, *Proceedings of the Twentieth Annual IEEE Symp. on Logic in Computer Science, LICS 2005.* IEEE Computer Society Press, June 2005. Short Presentation.

[16] Patrick Lincoln, John C. Mitchell, Andre Scedrov, and Natarajan Shankar. Decision problems for propositional linear logic. *Ann. Pure Appl. Logic*, 56(1-3):239–311, 1992.

[17] John Maraist, Martin Odersky, David N. Turner, and Philip Wadler. Call-by-name, call-by-value, call-by-need, and the linear lambda calculus. In *11th International Conference on the Mathematical Foundations of Programming Semantics*, 1995.

[18] Greg Morrisett, Amal J. Ahmed, and Matthew Fluet. L^3: A linear language with locations. In *TLCA*, pages 293–307, 2005.

[19] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, volume 21, pages 527–568. ACM Press, 1999.

[20] Benjamin C. Pierce. *Programming with Intersection Types and Bounded Polymorphism.* PhD thesis, 1991.

[21] Benjamin C. Pierce. *Types and Programming Languages.* The MIT Press, 2002.

[22] J. Reynolds. Separation logic: a logic for shared mutable data structures. In *3rd ACM SIGPLAN Workshop on Types in Compilation (TIC2000)*, 2002.

[23] Frederick Smith, David Walker, and Greg Morrisett. Alias types. Technical Report TR99-1773, Department of Computer Science, Cornell University, 1999.

[24] Frederick Smith, David Walker, and Greg Morrisett. Alias types. In *In European Symposium on Programming*, 2000.

[25] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[26] David N. Turner and Philip Wadler. Operational interpretations of linear logic. *Theoretical Computer Science*, 227(1–2):231–248, 1999.

[27] P. Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, pages 347–359. North Holland, 1990.

[28] P. L. Wadler. A taste of linear logic. In *Proceedings of the 18th International Symposium on Mathematical Foundations of Computer Science, Gdánsk*, New York, NY, 1993. Springer-Verlag.

[29] Philip Wadler. Is there a use for linear logic? In *Proceedings of the Symposium on Partial Evaluations and Semantics-Based Program Manipulation*, pages 255–273, New Haven, Connecticut, 1991.

[30] David Walker. Mechanical reasoning about low-level programs. lecture notes, http://www.cs.cmu.edu/~dpw/papers.html, 2001.

[31] David Walker and Kevin Watkins. On regions and linear types (extended abstract). In *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, pages 181–192. ACM Press, 2001.

[32] Lea Wittie. Type-Safe Operating System Abstractions. Technical Report TR2004-526, Dartmouth College, Computer Science, Hanover, NH, June 2004.

A Syntax and typing rules: CC0,CC1,CC2,LC

CC0, CC1, CC2, and LC are defined as extensions to a common sublanguage, CoreCC.

A.1 CoreCC

$$
\begin{array}{rcl}
\textit{kinds} \quad \kappa & = & \text{Type} \mid \text{Res} \mid \text{Cap} \\
\textit{constructors} \quad c & = & \alpha \mid \tau \mid C \\
\textit{ctor vars}\, \alpha, \beta, \epsilon, \rho, \ldots & & \\
\textit{types} \quad \tau & = & \alpha \mid \rho\,\text{handle} \mid \forall \alpha{:}\kappa.\tau \mid (C, \tau) \rightarrow 0 \mid \tau_1 \times \tau_2
\end{array}
$$

$$
\begin{array}{lcl}
\textit{capabilities} & C & = & \epsilon \mid \emptyset \\
\\
\textit{ctor ctxts} & \Delta & = & \cdot \mid \Delta, \alpha : \kappa \\
\textit{value ctxts} & \Gamma & = & \cdot \mid \Gamma, x : \tau \\
\textit{word values} & v & = & x \mid v[c : \kappa] \\
\textit{heap values} & h & = & \lambda\alpha{:}\kappa.h \mid \lambda(C, x : \tau).e \mid (v_1, v_2) \\
\textit{declarations} & d & = & x = v \mid x = h \mid x = \#n\, v \mid \text{new}\, \rho, x \mid \text{free}\, v \mid \text{use}\, v \\
\textit{expressions} & e & = & \text{let}\, d\, \text{in}\, e \mid v_1\ v_2 \mid \text{halt}
\end{array}
$$

$$\Delta \vdash \cdot \qquad \frac{\Delta \vdash \Delta'}{\Delta \vdash \Delta', \alpha : \kappa}(\alpha \notin \textit{domain}(\Delta, \Delta'))$$

$$\Delta \vdash \cdot \qquad \frac{\Delta \vdash \Gamma \qquad \Delta \vdash \tau : \text{Type}}{\Delta \vdash \Gamma, x : \tau}(x \notin \textit{domain}(\Gamma))$$

$$\ldots, \alpha : \kappa, \ldots \vdash \alpha : \kappa \qquad \frac{\Delta \vdash \alpha : \text{Res}}{\Delta \vdash \alpha\, \text{handle} : \text{Type}} \qquad \frac{\Delta \vdash \tau_1 : \text{Type} \qquad \Delta \vdash \tau_2 : \text{Type}}{\Delta \vdash \tau_1 \times \tau_2 : \text{Type}}$$

$$\frac{\Delta, \alpha : \kappa \vdash \tau : \text{Type}}{\Delta \vdash \forall\alpha{:}\kappa.\tau : \text{Type}}(\alpha \notin \textit{domain}(\Delta)) \qquad \frac{\Delta \vdash \tau : \text{Type} \qquad \Delta \vdash C : \text{Cap}}{\Delta \vdash (C, \tau) \to 0 : \text{Type}}$$

$$\frac{\Delta \vdash c : \kappa}{\Delta \vdash c = c : \kappa} \qquad \frac{\Delta \vdash c_2 = c_1 : \kappa}{\Delta \vdash c_1 = c_2 : \kappa} \qquad \frac{\Delta \vdash c_1 = c_2 : \kappa \qquad \Delta \vdash c_2 = c_3 : \kappa}{\Delta \vdash c_1 = c_3 : \kappa}$$

$$\frac{\Delta \vdash \tau_1 = \tau_1' : \text{Type} \qquad \Delta \vdash \tau_2 = \tau_2' : \text{Type}}{\Delta \vdash \tau_1 \times \tau_2 = \tau_1' \times \tau_2' : \text{Type}} \qquad \frac{\Delta, \alpha : \kappa \vdash \tau = \tau' : \text{Type}}{\Delta \vdash \forall\alpha{:}\kappa.\tau = \forall\alpha{:}\kappa.\tau' : \text{Type}}(\alpha \notin \textit{domain}(\Delta))$$

$$\frac{\Delta \vdash \tau = \tau' : \text{Type} \qquad \Delta \vdash C = C' : \text{Cap}}{\Delta \vdash (C, \tau) \to 0 = (C', \tau') \to 0 : \text{Type}}$$

$$\frac{\Delta, \alpha : \kappa; \Gamma \vdash h : \tau}{\Delta; \Gamma \vdash (\lambda\alpha{:}\kappa.h) : (\forall\alpha{:}\kappa.\tau)}(\alpha \notin \textit{domain}(\Delta))$$

$$\frac{\Delta \vdash C : \text{Cap} \qquad \Delta \vdash \tau : \text{Type} \qquad \Delta; \Gamma, x : \tau; C \vdash e}{\Delta; \Gamma \vdash \lambda(C, x : \tau).e : (C, \tau) \to 0}(x \notin \textit{domain}(\Gamma))$$

$$\frac{\Delta; \Gamma \vdash v_1 : \tau_1 \qquad \Delta; \Gamma \vdash v_2 : \tau_2}{\Delta; \Gamma \vdash (v_1, v_2) : \tau_1 \times \tau_2}$$

$$\frac{\Delta; \Gamma \vdash h : \tau' \qquad \Delta \vdash \tau' = \tau : \text{Type}}{\Delta; \Gamma \vdash h : \tau}$$

$$\Delta; \ldots, x : \tau, \ldots \vdash x : \tau$$

$$\frac{\Delta; \Gamma \vdash v : \forall\alpha{:}\kappa.\tau \qquad \Delta \vdash c : \kappa}{\Delta; \Gamma \vdash v[c : \kappa] : [\alpha \leftarrow c]\tau}$$

$$\frac{\Delta; \Gamma \vdash v : \tau' \qquad \Delta \vdash \tau' = \tau : \text{Type}}{\Delta; \Gamma \vdash v : \tau}$$

$$\frac{\Delta; \Gamma \vdash v : \tau}{\Delta; \Gamma; C \vdash x = v \Longrightarrow \Delta; \Gamma, x : \tau; C}(x \notin \mathit{domain}(\Gamma))$$

$$\frac{\Delta; \Gamma \vdash h : \tau}{\Delta; \Gamma; C \vdash x = h \Longrightarrow \Delta; \Gamma, x : \tau; C}(x \notin \mathit{domain}(\Gamma))$$

$$\frac{\Delta; \Gamma \vdash v : \tau_1 \times \tau_2}{\Delta; \Gamma; C \vdash x = \#n\, v \Longrightarrow \Delta; \Gamma, x : \tau_n; C}(x \notin \mathit{domain}(\Gamma)\ \mathit{and}\, n \in \{1, 2\})$$

$$\frac{\Delta; \Gamma; C \vdash d \Longrightarrow \Delta'; \Gamma'; C' \qquad \Delta'; \Gamma'; C' \vdash e}{\Delta; \Gamma; C \vdash \text{let}\, d \text{ in}\, e}$$

A.2 CC0

CC0 consists of CoreCC, plus the following.

$$\begin{array}{rcll}
\mathit{types} & \tau & = & \ldots \mid \forall \alpha \leq C.\tau \\
\mathit{capabilities} & C & = & \ldots \mid \{\rho^{\varphi}\} \mid C_1 \oplus C_2 \mid \overline{C} \\
\mathit{multiplicities} & \varphi & = & 1 \mid + \\
\mathit{ctor\, ctxts} & \Delta & = & \ldots \mid \Delta, \epsilon \leq C
\end{array}$$

$$\begin{array}{rcll}
\mathit{heap\, values} & h & = & \ldots \mid \lambda \alpha \leq C.h
\end{array}$$

$$\frac{\Delta \vdash \Delta' \qquad \Delta, \Delta' \vdash C : \text{Cap}}{\Delta \vdash \Delta', \alpha \leq C}(\alpha \notin \mathit{domain}(\Delta, \Delta'))$$

$$\frac{\Delta \vdash C : \text{Cap} \qquad \Delta, \alpha \leq C \vdash \tau : \text{Type}}{\Delta \vdash \forall \alpha \leq C.\tau : \text{Type}}(\alpha \notin \mathit{domain}(\Delta))$$

$$\ldots, \alpha \leq C, \ldots \vdash \alpha : \text{Cap} \qquad \Delta \vdash \emptyset : \text{Cap} \qquad \frac{\Delta \vdash \alpha : \text{Res}}{\Delta \vdash \{\alpha^{\varphi}\} : \text{Cap}}$$

$$\frac{\Delta \vdash C_1 : \kappa \qquad \Delta \vdash C_2 : \kappa}{\Delta \vdash C_1 \oplus C_2 : \kappa}(\kappa = \text{Cap}) \qquad \frac{\Delta \vdash C : \text{Cap}}{\Delta \vdash \overline{C} : \text{Cap}}$$

$$\frac{\Delta \vdash C_1 = C_1' : \kappa \qquad \Delta \vdash C_2 = C_2' : \kappa}{\Delta \vdash C_1 \oplus C_2 = C_1' \oplus C_2' : \kappa}(\kappa = \text{Cap})$$

$$\frac{\Delta \vdash C : \kappa}{\Delta \vdash \emptyset \oplus C = C : \kappa}(\kappa = \mathrm{Cap}) \qquad \frac{\Delta \vdash C_1 : \kappa \quad \Delta \vdash C_2 : \kappa}{\Delta \vdash C_1 \oplus C_2 = C_2 \oplus C_1 : \kappa}(\kappa = \mathrm{Cap})$$

$$\frac{\Delta \vdash C_1 : \kappa \quad \Delta \vdash C_2 : \kappa \quad \Delta \vdash C_3 : \kappa}{\Delta \vdash (C_1 \oplus C_2) \oplus C_3 = C_1 \oplus (C_2 \oplus C_3) : \kappa}(\kappa = \mathrm{Cap})$$

$$\frac{\Delta \vdash C = C' : \mathrm{Cap}}{\Delta \vdash \overline{C} = \overline{C'} : \mathrm{Cap}} \qquad \frac{\Delta \vdash C : \mathrm{Cap}}{\Delta \vdash \overline{C} = \overline{C} \oplus \overline{C} : \mathrm{Cap}}$$

$$\Delta \vdash \overline{\emptyset} = \emptyset : \mathrm{Cap} \qquad \frac{\Delta \vdash \alpha : \mathrm{Res}}{\Delta \vdash \overline{\{\alpha^1\}} = \{\alpha^+\} : \mathrm{Cap}}$$

$$\frac{\Delta \vdash C : \mathrm{Cap}}{\Delta \vdash \overline{\overline{C}} = \overline{C} : \mathrm{Cap}} \qquad \frac{\Delta \vdash C_1 : \mathrm{Cap} \quad \Delta \vdash C_2 : \mathrm{Cap}}{\Delta \vdash \overline{C_1 \oplus C_2} = \overline{C_1} \oplus \overline{C_2} : \mathrm{Cap}}$$

$$\frac{\Delta \vdash C_1 = C_2 : \kappa}{\Delta \vdash C_1 \leq C_2}(\kappa = \mathrm{Cap}) \qquad \frac{\Delta \vdash C_1 \leq C_2 \quad \Delta \vdash C_2 \leq C_3}{\Delta \vdash C_1 \leq C_3}$$

$$\frac{\Delta \vdash C_1 \leq C_1' \quad \Delta \vdash C_2 \leq C_2'}{\Delta \vdash C_1 \oplus C_2 \leq C_1' \oplus C_2'}$$

$$\frac{\Delta \vdash C \leq C'}{\Delta \vdash \overline{C} \leq \overline{C'}} \qquad \ldots, \alpha \leq C, \ldots \vdash \alpha \leq C \qquad \frac{\Delta \vdash C : \mathrm{Cap}}{\Delta \vdash C \leq \overline{C}}$$

$$\frac{\Delta \vdash C = C' : \mathrm{Cap} \quad \Delta, \alpha \leq C \vdash \tau = \tau' : \mathrm{Type}}{\Delta \vdash \forall \alpha \leq C.\tau = \forall \alpha \leq C'.\tau' : \mathrm{Type}}(\alpha \notin \mathit{domain}(\Delta))$$

$$\frac{\Delta \vdash C : \mathrm{Cap} \quad \Delta, \alpha \leq C; \Gamma \vdash h : \tau}{\Delta; \Gamma \vdash \lambda \alpha \leq C.h : \forall \alpha \leq C.\tau}(\alpha \notin \mathit{domain}(\Delta))$$

$$\frac{\Delta; \Gamma \vdash v : \forall \alpha \leq C'.\tau \quad \Delta \vdash C \leq C'}{\Delta; \Gamma \vdash v[C : \mathrm{Cap}] : [\alpha \leftarrow C]\tau}$$

$$\Delta; \Gamma; C \vdash \mathrm{new}\, \alpha, x \Longrightarrow \Delta, \alpha : \mathrm{Res}; \Gamma, x : \alpha\, \mathrm{handle}; C \oplus \{\alpha^1\} \qquad (\alpha \notin \mathit{domain}(\Delta)\ \mathit{and}\ x \notin \mathit{domain}(\Gamma))$$

$$\frac{\Delta; \Gamma \vdash v : \alpha\, \mathrm{handle} \quad \Delta \vdash C = C' \oplus \{\alpha^1\} : \mathrm{Cap}}{\Delta; \Gamma; C \vdash \mathrm{free}\, v \Longrightarrow \Delta; \Gamma; C'}$$

$$\frac{\Delta; \Gamma \vdash v : \alpha\, \mathrm{handle} \quad \Delta \vdash C \leq C' \oplus \{\alpha^+\}}{\Delta; \Gamma; C \vdash \mathrm{use}\, v \Longrightarrow \Delta; \Gamma; C}$$

$$\frac{\Delta; \Gamma \vdash v_1 : (C', \tau) \rightarrow 0 \quad \Delta; \Gamma \vdash v_2 : \tau \quad \Delta \vdash C \leq C'}{\Delta; \Gamma; C \vdash v_1\ v_2}$$

$$\frac{\Delta \vdash C = \emptyset : \mathrm{Cap}}{\Delta; \Gamma; C \vdash \mathrm{halt}}$$

A.3 CC1

CC1 consists of CoreCC, plus the following.

$$
\begin{array}{rrcl}
\textit{kinds} & \kappa & = & \ldots \mid \text{Cap}^+ \\
\\
\textit{types} & \tau & = & \ldots \mid \forall\alpha{:}\text{Cap}^+ \le C.\tau \mid \forall\alpha{:}\text{Cap} \le C_0, C_1, \ldots, C_n.\tau \\
\textit{capabilities} & C & = & \ldots \mid \{\rho^\varphi\} \mid C_1 \oplus C_2 \\
\textit{multiplicities} & \varphi & = & 1 \mid + \\
\textit{ctor ctxts} & \Delta & = & \ldots \mid \Delta, \epsilon : \text{Cap}^+ \le C \mid \Delta, \epsilon : \text{Cap} \le (C_0, C_1, \ldots, C_n) \\
\\
\textit{heap values} & h & = & \ldots \mid \lambda\alpha{:}\text{Cap}^+ \le C.h \mid \lambda\alpha{:}\text{Cap} \le C_0, C_1, \ldots, C_n.h
\end{array}
$$

$$\frac{\Delta \vdash \Delta' \qquad \Delta, \Delta' \vdash C : \text{Cap}^+}{\Delta \vdash \Delta', \alpha : \text{Cap}^+ \le C}(\alpha \notin \textit{domain}(\Delta, \Delta'))$$

$$\frac{\Delta \vdash \Delta' \qquad \Delta, \Delta' \vdash C_0 : \text{Cap} \qquad \Delta, \Delta' \vdash C_1 : \text{Cap}^+ \qquad \ldots \qquad \Delta, \Delta' \vdash C_n : \text{Cap}^+}{\Delta \vdash \Delta', \alpha : \text{Cap} \le (C_0, C_1, \ldots, C_n)}(\alpha \notin \textit{domain}(\Delta, \Delta'))$$

$$\ldots, \alpha : \text{Cap}^+ \le C, \ldots \vdash \alpha : \text{Cap}^+$$

$$\ldots, \alpha : \text{Cap} \le (C_0, C_1, \ldots, C_n), \ldots \vdash \alpha : \text{Cap}$$

$$\frac{\Delta \vdash C : \text{Cap}^+ \qquad \Delta, \alpha : \text{Cap}^+ \le C \vdash \tau : \text{Type}}{\Delta \vdash \forall\alpha{:}\text{Cap}^+ \le C.\tau : \text{Type}}(\alpha \notin \textit{domain}(\Delta))$$

$$\frac{\begin{array}{c}\Delta \vdash C_0 : \text{Cap} \qquad \Delta \vdash C_1 : \text{Cap}^+ \qquad \ldots \qquad \Delta \vdash C_n : \text{Cap}^+ \\ \Delta, \alpha : \text{Cap} \le (C_0, C_1, \ldots, C_n) \vdash \tau : \text{Type}\end{array}}{\Delta \vdash \forall\alpha{:}\text{Cap} \le C_0, C_1, \ldots, C_n.\tau : \text{Type}}(\alpha \notin \textit{domain}(\Delta))$$

$$\frac{\Delta \vdash \alpha : \text{Res}}{\Delta \vdash \{\alpha^1\} : \text{Cap}} \qquad \frac{\Delta \vdash \alpha : \text{Res}}{\Delta \vdash \{\alpha^+\} : \text{Cap}^+} \qquad \frac{\Delta \vdash C : \text{Cap}^+}{\Delta \vdash C : \text{Cap}}$$

$$\Delta \vdash \emptyset : \text{Cap}^+ \qquad \frac{\Delta \vdash C_1 : \kappa \qquad \Delta \vdash C_2 : \kappa}{\Delta \vdash C_1 \oplus C_2 : \kappa}(\kappa \in \{\text{Cap}, \text{Cap}^+\})$$

$$\frac{\Delta \vdash C_1 = C_1' : \kappa \qquad \Delta \vdash C_2 = C_2' : \kappa}{\Delta \vdash C_1 \oplus C_2 = C_1' \oplus C_2' : \kappa}(\kappa \in \{\text{Cap}, \text{Cap}^+\})$$

$$\frac{\Delta \vdash C : \kappa}{\Delta \vdash \emptyset \oplus C = C : \kappa}(\kappa \in \{\text{Cap}, \text{Cap}^+\}) \qquad \frac{\Delta \vdash C_1 : \kappa \qquad \Delta \vdash C_2 : \kappa}{\Delta \vdash C_1 \oplus C_2 = C_2 \oplus C_1 : \kappa}(\kappa \in \{\text{Cap}, \text{Cap}^+\})$$

$$\frac{\Delta \vdash C_1 : \kappa \qquad \Delta \vdash C_2 : \kappa \qquad \Delta \vdash C_3 : \kappa}{\Delta \vdash (C_1 \oplus C_2) \oplus C_3 = C_1 \oplus (C_2 \oplus C_3) : \kappa} (\kappa \in \{\mathrm{Cap}, \mathrm{Cap}^+\})$$

$$\frac{\Delta \vdash C_1 = C_2 : \mathrm{Cap}^+}{\Delta \vdash C_1 = C_2 : \mathrm{Cap}}$$

$$\frac{\Delta \vdash C : \mathrm{Cap}^+}{\Delta \vdash C = C \oplus C : \mathrm{Cap}^+}$$

$$\frac{\Delta \vdash C_1 = C_2 : \kappa}{\Delta \vdash C_1 \leq C_2} (\kappa \in \{\mathrm{Cap}, \mathrm{Cap}^+\}) \qquad \frac{\Delta \vdash C_1 \leq C_2 \qquad \Delta \vdash C_2 \leq C_3}{\Delta \vdash C_1 \leq C_3}$$

$$\frac{\Delta \vdash C_1 \leq C_1' \qquad \Delta \vdash C_2 \leq C_2'}{\Delta \vdash C_1 \oplus C_2 \leq C_1' \oplus C_2'}$$

$$\ldots, \alpha : \kappa \leq (\ldots, C, \ldots), \ldots \vdash \alpha \leq C$$

$$\frac{\Delta \vdash \alpha : \mathrm{Res}}{\Delta \vdash \{\alpha^1\} \leq \{\alpha^+\}}$$

$$\frac{\Delta \vdash C = C' : \mathrm{Cap}^+ \qquad \Delta, \alpha : \mathrm{Cap}^+ \leq C \vdash \tau = \tau' : \mathrm{Type}}{\Delta \vdash \forall \alpha{:}\mathrm{Cap}^+ \leq C.\tau = \forall \alpha{:}\mathrm{Cap}^+ \leq C'.\tau' : \mathrm{Type}} (\alpha \notin domain(\Delta))$$

$$\frac{\begin{array}{c}\Delta \vdash C_0 = C_0' : \mathrm{Cap} \qquad \Delta \vdash C_1 = C_1' : \mathrm{Cap}^+ \qquad \ldots \qquad \Delta \vdash C_n = C_n' : \mathrm{Cap}^+ \\ \Delta, \alpha : \mathrm{Cap} \leq (C_0, C_1, \ldots, C_n) \vdash \tau = \tau' : \mathrm{Type}\end{array}}{\Delta \vdash \forall \alpha{:}\mathrm{Cap} \leq (C_0, C_1, \ldots, C_n).\tau = \forall \alpha{:}\mathrm{Cap} \leq (C_0', C_1', \ldots, C_n').\tau' : \mathrm{Type}} (\alpha \notin domain(\Delta))$$

$$\frac{\Delta \vdash \forall \alpha{:}\kappa \leq C_1, \ldots, C_n.\tau : \mathrm{Type} \qquad \Delta, \alpha \leq (C_1, \ldots, C_n); \Gamma \vdash h : \tau}{\Delta; \Gamma \vdash (\lambda \alpha{:}\kappa \leq C_1, \ldots, C_n.h) : (\forall \alpha{:}\kappa \leq C_1, \ldots, C_n.\tau)} (\alpha \notin domain(\Delta))$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash v : \forall \alpha{:}\kappa \leq C_1, \ldots, C_n.\tau \\ \Delta \vdash C \leq C_1 \qquad \Delta \vdash C \leq C_n \\ \Delta \vdash C : \kappa\end{array}}{\Delta; \Gamma \vdash v[C : \kappa] : [\alpha \leftarrow C]\tau} (\kappa \in \{\mathrm{Cap}, \mathrm{Cap}^+\})$$

Use CC0's typing rules for: $(\mathrm{new}\, \alpha, x)$, $(\mathrm{free}\, v)$, $(\mathrm{use}\, v)$, $(v_1\ v_2)$, (halt).

A.4 CC2

CC2 consists of CoreCC, plus the following.

$$\begin{array}{rcll}
\textit{kinds} & \kappa & = & \mathrm{Type} \mid \mathrm{Res} \mid \mathrm{Cap}^{\varphi} \\
\textit{constructors} & c & = & \alpha \mid \tau \mid Q \\
\\
\textit{types} & \tau & = & \ldots \mid \forall \alpha{:}\mathrm{Cap}^+ \,\mathrm{where}\, U_1 \boxplus \alpha \leq A_1, \ldots, U_n \boxplus \alpha \leq A_n.\tau \\
\textit{pure capabilities} & Q, A, U & = & \alpha \mid \emptyset \mid \{\rho^{\varphi}\} \mid Q_1 \oplus Q_2 \\
\textit{mixed capabilities} & C & = & U \boxplus A \\
\textit{multiplicities} & \varphi & = & 1 \mid +
\end{array}$$

$$\textit{ctor ctxts} \quad \Delta \quad = \quad \ldots \mid \Delta, \alpha{:}\mathrm{Cap}^{+} \,\mathrm{where}\, U_1 \boxplus \alpha \le A_1, \ldots, U_n \boxplus \alpha \le A_n$$

$$\textit{heap values} \quad h \quad = \quad \ldots \mid \lambda\alpha{:}\mathrm{Cap}^{+} \,\mathrm{where}\, U_1 \boxplus \alpha \le A_1, \ldots, U_n \boxplus \alpha \le A_n.h$$

Let "$\Delta \vdash U \boxplus A : \mathrm{Cap}$" be an abbreviation for "$\Delta \vdash U : \mathrm{Cap}^1$ and $\Delta \vdash A : \mathrm{Cap}^{+}$".
Let "$\Delta \vdash U \boxplus A = U' \boxplus A' : \mathrm{Cap}$" be an abbreviation for "$\Delta \vdash U = U' : \mathrm{Cap}^1$ and $\Delta \vdash A = A' : \mathrm{Cap}^{+}$".

$$\frac{\begin{array}{c}\Delta \vdash \Delta' \\ \Delta, \Delta' \vdash U_1 : \mathrm{Cap}^1 \quad \ldots \quad \Delta, \Delta' \vdash U_n : \mathrm{Cap}^1 \\ \Delta, \Delta' \vdash A_1 : \mathrm{Cap}^{+} \quad \ldots \quad \Delta, \Delta' \vdash A_n : \mathrm{Cap}^{+}\end{array}}{\Delta \vdash \Delta', \alpha{:}\mathrm{Cap}^{+} \,\mathrm{where}\, U_1 \boxplus \alpha \le A_1, \ldots, U_n \boxplus \alpha \le A_n}(\alpha \notin \mathit{domain}(\Delta, \Delta'))$$

$$\ldots, \alpha{:}\mathrm{Cap}^{+} \,\mathrm{where}\, U_1 \boxplus \alpha \le A_1, \ldots, U_n \boxplus \alpha \le A_n, \ldots \vdash \alpha : \mathrm{Cap}^{+}$$

$$\frac{\begin{array}{c}\Delta \vdash U_1 : \mathrm{Cap}^1 \quad \ldots \quad \Delta \vdash U_n : \mathrm{Cap}^1 \\ \Delta \vdash A_1 : \mathrm{Cap}^{+} \quad \ldots \quad \Delta \vdash A_n : \mathrm{Cap}^{+} \\ \Delta, \alpha{:}\mathrm{Cap}^{+} \,\mathrm{where}\, U_1 \boxplus \alpha \le A_1, \ldots, U_n \boxplus \alpha \le A_n \vdash \tau : \mathrm{Type}\end{array}}{\Delta \vdash \forall\alpha{:}\mathrm{Cap}^{+} \,\mathrm{where}\, U_1 \boxplus \alpha \le A_1, \ldots, U_n \boxplus \alpha \le A_n.\tau : \mathrm{Type}}(\alpha \notin \mathit{domain}(\Delta))$$

$$\Delta \vdash \emptyset : \mathrm{Cap}^{\varphi} \qquad \frac{\Delta \vdash \alpha : \mathrm{Res}}{\Delta \vdash \{\alpha^{\varphi}\} : \mathrm{Cap}^{\varphi}} \qquad \frac{\Delta \vdash Q_1 : \kappa \quad \Delta \vdash Q_2 : \kappa}{\Delta \vdash Q_1 \oplus Q_2 : \kappa}(\kappa = \mathrm{Cap}^{\varphi})$$

$$\frac{\Delta \vdash Q_1 = Q_1' : \kappa \quad \Delta \vdash Q_2 = Q_2' : \kappa}{\Delta \vdash Q_1 \oplus Q_2 = Q_1' \oplus Q_2' : \kappa}(\kappa = \mathrm{Cap}^{\varphi})$$

$$\frac{\Delta \vdash Q : \kappa}{\Delta \vdash \emptyset \oplus Q = Q : \kappa}(\kappa = \mathrm{Cap}^{\varphi}) \qquad \frac{\Delta \vdash Q_1 : \kappa \quad \Delta \vdash Q_2 : \kappa}{\Delta \vdash Q_1 \oplus Q_2 = Q_2 \oplus Q_1 : \kappa}(\kappa = \mathrm{Cap}^{\varphi})$$

$$\frac{\Delta \vdash Q_1 : \kappa \quad \Delta \vdash Q_2 : \kappa \quad \Delta \vdash Q_3 : \kappa}{\Delta \vdash (Q_1 \oplus Q_2) \oplus Q_3 = Q_1 \oplus (Q_2 \oplus Q_3) : \kappa}(\kappa = \mathrm{Cap}^{\varphi})$$

$$\frac{\Delta \vdash A : \mathrm{Cap}^{+}}{\Delta \vdash A = A \oplus A : \mathrm{Cap}^{+}}$$

$$\ldots, (\alpha{:}\mathrm{Cap}^{+} \,\mathrm{where}\, \ldots, U \boxplus \alpha \le A, \ldots), \ldots \vdash U \boxplus \alpha \le A$$

$$\frac{\Delta \vdash \alpha : \mathrm{Res}}{\Delta \vdash \{\alpha^1\} \boxplus \emptyset \le \{\alpha^{+}\}} \qquad \frac{\Delta \vdash A_1 = A_2 : \mathrm{Cap}^{+}}{\Delta \vdash \emptyset \boxplus A_1 \le A_2}$$

$$\frac{\Delta \vdash U_1 \boxplus A_1 \le A_2 \quad \Delta \vdash U_2 \boxplus A_2 \le A_3}{\Delta \vdash U_1 \oplus U_2 \boxplus A_1 \le A_3}$$

$$\frac{\Delta \vdash U_1 \boxplus A_1 \le A_1' \quad \Delta \vdash U_2 \boxplus A_2 \le A_2'}{\Delta \vdash U_1 \oplus U_2 \boxplus A_1 \oplus A_2 \le A_1' \oplus A_2'}$$

$$\frac{\begin{array}{c}\Delta \vdash U_1 = U_1' : \mathrm{Cap}^1 \quad \ldots \quad \Delta \vdash U_n = U_n' : \mathrm{Cap}^1 \\ \Delta \vdash A_1 = A_1' : \mathrm{Cap}^+ \quad \ldots \quad \Delta \vdash A_n = A_n' : \mathrm{Cap}^+ \\ \Delta, \alpha{:}\mathrm{Cap}^+ \,\mathrm{where}\, U_1 \boxplus \alpha \leq A_1, \ldots, U_n \boxplus \alpha \leq A_n \vdash \tau = \tau' : \mathrm{Type}\end{array}}{\begin{array}{c}\Delta \vdash \forall\alpha{:}\mathrm{Cap}^+ \,\mathrm{where}\, U_1 \boxplus \alpha \leq A_1, \ldots, U_n \boxplus \alpha \leq A_n.\tau \\ = \forall\alpha{:}\mathrm{Cap}^+ \,\mathrm{where}\, U_1' \boxplus \alpha \leq A_1', \ldots, U_n' \boxplus \alpha \leq A_n'.\tau' : \mathrm{Type}\end{array}} (\alpha \notin \mathit{domain}(\Delta))$$

$$\frac{\begin{array}{c}\Delta \vdash \forall\alpha{:}\mathrm{Cap}^+ \,\mathrm{where}\, U_1 \boxplus \alpha \leq A_1, \ldots, U_n \boxplus \alpha \leq A_n.\tau : \mathrm{Type} \\ \Delta, \alpha{:}\mathrm{Cap}^+ \,\mathrm{where}\, U_1 \boxplus \alpha \leq A_1, \ldots, U_n \boxplus \alpha \leq A_n; \Gamma \vdash h : \tau\end{array}}{\begin{array}{c}\Delta; \Gamma \vdash \lambda\alpha{:}\mathrm{Cap}^+ \,\mathrm{where}\, U_1 \boxplus \alpha \leq A_1, \ldots, U_n \boxplus \alpha \leq A_n.h : \\ \forall\alpha{:}\mathrm{Cap}^+ \,\mathrm{where}\, U_1 \boxplus \alpha \leq A_1, \ldots, U_n \boxplus \alpha \leq A_n.\tau\end{array}} (\alpha \notin \mathit{domain}(\Delta))$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash v : \forall\alpha{:}\mathrm{Cap}^+ \,\mathrm{where}\, U_1 \boxplus \alpha \leq A_1, \ldots, U_n \boxplus \alpha \leq A_n.\tau \\ \Delta \vdash U_1 \boxplus \alpha \leq A_1 \quad \ldots \quad \Delta \vdash U_n \boxplus \alpha \leq A_n \\ \Delta; \Gamma \vdash A : \mathrm{Cap}^+\end{array}}{\Delta; \Gamma \vdash v[A : \mathrm{Cap}^+] : [\alpha \leftarrow A]\tau}$$

$$\Delta; \Gamma; U \boxplus A \vdash \mathrm{new}\, \alpha, x \Longrightarrow \Delta, \alpha : \mathrm{Res}; \Gamma, x : \alpha\, \mathrm{handle}; U \oplus \{\alpha^1\} \boxplus A \qquad (\alpha \notin \mathit{domain}(\Delta)\, \mathit{and}\, x \notin \mathit{domain}(\Gamma))$$

$$\frac{\Delta; \Gamma \vdash v : \alpha\, \mathrm{handle} \qquad \Delta \vdash U = U' \oplus \{\alpha^1\} : \mathrm{Cap}^1}{\Delta; \Gamma; U \boxplus A \vdash \mathrm{free}\, v \Longrightarrow \Delta; \Gamma; U' \boxplus A}$$

$$\frac{\Delta; \Gamma \vdash v : \alpha\, \mathrm{handle} \qquad \Delta \vdash U = U_B \oplus U' : \mathrm{Cap}^1 \qquad \Delta \vdash U_B \boxplus A \leq A' \oplus \{\alpha^+\}}{\Delta; \Gamma; U \boxplus A \vdash \mathrm{use}\, v \Longrightarrow \Delta; \Gamma; U \boxplus A}$$

$$\frac{\Delta; \Gamma \vdash v_1 : (U' \boxplus A', \tau) \rightarrow 0 \qquad \Delta; \Gamma \vdash v_2 : \tau \qquad \Delta \vdash U = U_B \oplus U' : \mathrm{Cap}^1 \qquad \Delta \vdash U_B \boxplus A \leq A'}{\Delta; \Gamma; U \boxplus A \vdash v_1\ v_2}$$

$$\frac{\Delta \vdash U = \emptyset : \mathrm{Cap}^1 \qquad \Delta \vdash A = \emptyset : \mathrm{Cap}^+}{\Delta; \Gamma; U \boxplus A \vdash \mathrm{halt}}$$

A.5 LC

LC consists of CoreCC, plus the following.

$$\begin{array}{rcl}\mathit{capabilities} & C & = & \ldots \mid \{\rho\} \mid C_1 \otimes C_2 \mid C_1 \& C_2 \mid C_1 \multimap C_2 \mid \mathrm{true} \\ \mathit{cap\, ctxts} & \Lambda & = & C_1, \ldots, C_n\end{array}$$

$$\Delta \vdash \emptyset : \mathrm{Cap} \qquad \frac{\Delta \vdash \alpha : \mathrm{Res}}{\Delta \vdash \{\alpha\} : \mathrm{Cap}} \qquad \Delta \vdash \mathrm{true} : \mathrm{Cap}$$

$$\frac{\Delta \vdash C_1 : \mathrm{Cap} \qquad \Delta \vdash C_2 : \mathrm{Cap}}{\Delta \vdash C_1 \otimes C_2 : \mathrm{Cap}} \qquad \frac{\Delta \vdash C_1 : \mathrm{Cap} \qquad \Delta \vdash C_2 : \mathrm{Cap}}{\Delta \vdash C_1 \& C_2 : \mathrm{Cap}}$$

$$\frac{\Delta \vdash C_1 : \text{Cap} \qquad \Delta \vdash C_2 : \text{Cap}}{\Delta \vdash C_1 \multimap C_2 : \text{Cap}}$$

$$C \vdash C \qquad\qquad \vdash \emptyset \qquad\qquad \frac{\Lambda \vdash C}{\Lambda, \emptyset \vdash C} \qquad\qquad \Lambda \vdash \text{true}$$

$$\frac{\Lambda_1 \vdash C_1 \qquad \Lambda_2 \vdash C_2}{\Lambda_1, \Lambda_2 \vdash C_1 \otimes C_2} \qquad\qquad \frac{\Lambda \vdash C_1 \qquad \Lambda \vdash C_2}{\Lambda \vdash C_1 \& C_2} \qquad\qquad \frac{\Lambda, C_1 \vdash C_2}{\Lambda \vdash C_1 \multimap C_2}$$

$$\frac{\Lambda, C_1, C_2 \vdash C_3}{\Lambda, C_1 \otimes C_2 \vdash C_3} \qquad\qquad \frac{\Lambda, C_k \vdash C_3}{\Lambda, C_1 \& C_2 \vdash C_3}(k \in \{1,2\}) \qquad\qquad \frac{\Lambda_1 \vdash C_1 \qquad \Lambda_2, C_2 \vdash C_3}{\Lambda_1, \Lambda_2, C_1 \multimap C_2 \vdash C_3}$$

$$\frac{\Delta \vdash C_1 : \kappa \qquad \Delta \vdash C_2 : \kappa \qquad C_1 \vdash C_2 \qquad C_2 \vdash C_1}{\Delta \vdash C_1 = C_2 : \kappa}$$

$$\frac{C \otimes \{\alpha\} \vdash C'}{\Delta; \Gamma; C \vdash \text{new}\, \alpha, x \Longrightarrow \Delta, \alpha : \text{Res}; \Gamma, x : \alpha\, \text{handle}; C'}(\alpha \notin \mathit{domain}(\Delta)\, \mathit{and}\, x \notin \mathit{domain}(\Gamma))$$

$$\frac{\Delta; \Gamma \vdash v : \alpha\, \text{handle} \qquad C \vdash C' \otimes \{\alpha\}}{\Delta; \Gamma; C \vdash \text{free}\, v \Longrightarrow \Delta; \Gamma; C'}$$

$$\frac{\Delta; \Gamma \vdash v : \alpha\, \text{handle} \qquad C \vdash \{\alpha\} \otimes \text{true}}{\Delta; \Gamma; C \vdash \text{use}\, v \Longrightarrow \Delta; \Gamma; C}$$

$$\frac{\Delta; \Gamma \vdash v_1 : (C', \tau) \to 0 \qquad \Delta; \Gamma \vdash v_2 : \tau \qquad C \vdash C'}{\Delta; \Gamma; C \vdash v_1\ v_2}$$

$$\Delta; \Gamma; C \vdash \text{halt}$$

B Linear $F\omega$

Linear $F\omega$ extends standard $F\omega$[21] with linear types. The syntax is straightforward:

$$\begin{array}{rcll}
\mathit{kinds} & \kappa & = & \text{Type} \mid \kappa \to \kappa \\
\mathit{types} & \tau & = & \alpha \mid \forall\alpha{:}\kappa.\tau \mid \tau_1 \multimap \tau_2 \mid\ !\tau \mid \lambda\alpha{:}\kappa.\tau \mid \tau_1\ \tau_2 \\
\mathit{expressions} & e & = & x \mid \lambda\alpha{:}\kappa.e \mid e\ \tau \mid \lambda(\phi x){:}\tau.e \mid e_1\ e_2 \mid\ !e \\
\mathit{linearities} & \phi & = & \cdot \mid\ ! \\
\mathit{type\,ctxts} & \Delta & = & \cdot \mid \Delta, \alpha : \kappa \\
\mathit{value\,ctxts} & \Gamma & = & \cdot \mid \Gamma, \phi(x : \tau)
\end{array}$$

The typing rules require some explanation, because obvious typing rules for the $!e$ expression lead to subtle problems. First, unrestricted use of $!e$ would allows a program to copy linear resources – if variable x has linear type $file$, then the expression $!x$ would have nonlinear type $!file$, which allows unrestricted duplication of a file handle. More subtly, the nonlinear function $!\lambda y{:}().x$ has type $!(() \multimap file)$, so the program may duplicate the function and then

call each copy of the function separately to obtain multiple copies of the same file handle; in this case, each file copy has linear type $file$, giving no indication that other copies of the file exist. The standard solution to this problem is to restrict the typing rules so that $!e$ only type-checks in a nonlinear environment $!\Gamma$, which contains no linear assumptions. This rules out both the $!x$ and $!\lambda y\!:\!().x$ examples, since $x : file$ is a linear assumption.

Unfortunately, this isn't the end of the story, because many systems have nonlinear functions that create linear resources. Consider a nonlinear function *open* of type $!(string \multimap file)$. Even with the nonlinear environment restriction described above, the expression $!(open\ \text{"}foo.c\text{"})$ is legal and has nonlinear type $!file$. One solution to this, at least in a call-by-value system, is to restrict the expression e in $!e$ to be a value [18] (this is occasionally inconvenient — the nonlinear pair $!\langle 2+2, 3\rangle$ must be written as $\text{let}\, x = 2+2 \text{ in } !\langle x, 3\rangle$) . The rules below follow Wadler's "steadfast types" [29][27], which limit the typing rules for $!e$ to particular forms of e, including functions $!\lambda x\!:\!\tau.e$ and pairs $!\langle e_1, e_2\rangle$, but not variables $!x$ and function applications $!(e_1\ e_2)$.

In addition to the problem with expressions creating linear resources, a non-call-by-value linear system must address the reverse problem: expressions consuming linear resources. Consider a nonlinear function *close* of type $!(file \multimap ())$. Assuming x has linear type $file$, the expression $(close\ x)$ has nonlinear type $()$ and may therefore be copied, as in the expression $(\lambda z\!:\!().\,!\langle z, z\rangle)\ (close\ x)$. A non-call-by-value language could copy the expression first and evaluate the copied expressions later, thereby closing the linear file x more than once. Furthermore, the unevaluated copied expressions will be ill-typed, because they all rely on a single linear variable x, which cannot be copied. The standard solution to this (see [28]) restricts the operational semantics so that any expression substituted for a nonlinear variable (such as $(close\ x)$ for z) must first be evaluated to a form $!e$, which is freely duplicable because, as described above, it type-checks using only nonlinear assumptions. For example, the evaluation rule for function calls, assuming z is nonlinear, would be:

$(\lambda z\!:\!\tau.e_b)\ (!e_a) \longrightarrow [z \leftarrow !e_a]e_b$

How does the evaluation rule know whether z is nonlinear? It could examine τ for nonlinearity, but a cleaner solution [28] is to distinguish between *linear variable bindings* λz and *nonlinear variable bindings* $\lambda !z$. A nonlinear parameter binding signals that a function argument must have the form $!e$ before substitution:

$(\lambda z\!:\!\tau.e_b)\ (e_a) \longrightarrow [z \leftarrow e_a]e_b$

$(\lambda\, !z\!:\!\tau.e_b)\ (!e_a) \longrightarrow [z \leftarrow !e_a]e_b$

(Note that [28] actually substitutes e_a for z, while the rule above substitutes $!e_a$ for z. In a non-steadfast type system, e_a is more useful than $!e_a$, and the program can easily recover $!e_a$ after the substitution by using the expression $!z$. Steadfast types prohibit the expression $!z$, though, making $!e_a$ is more useful than e_a.)

The environment Γ tracks linearly bound variables using assumptions of the form $x : \tau$, and nonlinearly bound variables using assumptions of the form $!(x : \tau)$. A nonlinear environment $!\Gamma$ contains only nonlinearly bound variables.

The environments $\Delta, \alpha : \kappa$ and $\Gamma, x : \tau$ are well-formed only if $\alpha \notin domain(\Delta)$ and $x \notin domain(\Gamma)$, respectively. The rules below apply only to well formed environments.

$$\{\ldots, \alpha : \kappa, \ldots\} \vdash \alpha : \kappa \qquad \frac{\Delta, \alpha : \kappa \vdash \tau : \text{Type}}{\Delta \vdash \forall \alpha\!:\!\kappa.\tau : \text{Type}} \qquad \frac{\Delta \vdash \tau_1 : \text{Type} \qquad \Delta \vdash \tau_2 : \text{Type}}{\Delta \vdash \tau_1 \multimap \tau_2 : \text{Type}} \qquad \frac{\Delta \vdash \tau : \text{Type}}{\Delta \vdash !\tau : \text{Type}}$$

$$\frac{\Delta, \alpha : \kappa_1 \vdash \tau : \kappa_2}{\Delta \vdash (\lambda \alpha\!:\!\kappa_1.\tau) : \kappa_1 \to \kappa_2} \qquad \frac{\Delta \vdash \tau_1 : \kappa_a \to \kappa_b \qquad \Delta \vdash \tau_2 : \kappa_a}{\Delta \vdash \tau_1\ \tau_2 : \kappa_b}$$

$$\frac{\Delta \vdash \tau}{\Delta \vdash \tau = \tau} \qquad \frac{\Delta \vdash \tau_2 = \tau_1}{\Delta \vdash \tau_1 = \tau_2} \qquad \frac{\Delta \vdash \tau_1 = \tau_2 \qquad \Delta \vdash \tau_2 = \tau_3}{\Delta \vdash \tau_1 = \tau_3}$$

$$\Delta \vdash (\lambda \alpha\!:\!\kappa.\tau_1)\ \tau_2 = [\alpha \leftarrow \tau_2]\tau_1$$

$$\frac{\Delta \vdash \tau = \tau'}{\Delta \vdash !\tau = !\tau'} \qquad \frac{\Delta \vdash \tau = \tau'}{\Delta \vdash \forall\alpha{:}\kappa.\tau = \forall\alpha{:}\kappa.\tau'} \qquad \frac{\Delta \vdash \tau_1 = \tau_1' \qquad \Delta \vdash \tau_2 = \tau_2'}{\Delta \vdash \tau_1 \multimap \tau_2 = \tau_1' \multimap \tau_2'}$$

$$\frac{\Delta \vdash \tau = \tau'}{\Delta \vdash \lambda\alpha{:}\kappa.\tau = \lambda\alpha{:}\kappa.\tau'} \qquad \frac{\Delta \vdash \tau_1 = \tau_1' \qquad \Delta \vdash \tau_2 = \tau_2'}{\Delta \vdash \tau_1\ \tau_2 = \tau_1'\ \tau_2'}$$

$$\Delta;\ !\Gamma, \phi(x : \tau) \vdash x : \phi\tau \qquad \frac{\Delta, \alpha : \kappa; \phi\Gamma \vdash e : \tau}{\Delta; \phi\Gamma \vdash (\phi\lambda\alpha{:}\kappa.e) : \phi\forall\alpha{:}\kappa.\tau} \qquad \frac{\Delta; \Gamma \vdash e : \phi\forall\alpha{:}\kappa.\tau_b \qquad \Delta \vdash \tau_a : \kappa}{\Delta; \Gamma \vdash e\ \tau_a : [\alpha \leftarrow \tau_a]\tau_b}$$

$$\frac{\Delta; (\phi'\Gamma), \phi(x : \tau_a) \vdash e : \tau_b}{\Delta; \phi'\Gamma \vdash (\phi'\lambda(\phi x){:}\phi\tau_a.e) : \phi'(\phi\tau_a \multimap \tau_b)} \qquad \frac{\Delta; \Gamma_1 \vdash e_1 : \phi(\tau_a \multimap \tau_b) \qquad \Delta; \Gamma_2 \vdash e_2 : \tau_a}{\Delta; \Gamma_1, \Gamma_2 \vdash (e_1\ e_2) : \tau_b}$$

$$\frac{\Delta; \Gamma \vdash e : \tau' \qquad \Delta \vdash \tau' = \tau}{\Delta; \Gamma \vdash e : \tau}$$

$$(\phi\lambda\alpha{:}\kappa.e)\ \tau \longrightarrow [\alpha \leftarrow \tau]e \qquad (\phi'\lambda(\phi x){:}\tau.e_b)\ (\phi e_a) \longrightarrow [x \leftarrow \phi e_a]e_b$$

$$\frac{e \longrightarrow e'}{e\ \tau \longrightarrow e'\ \tau} \qquad \frac{e \longrightarrow e'}{e\ e_2 \longrightarrow e'\ e_2} \qquad \frac{e \longrightarrow e'}{e_1\ e \longrightarrow e_1\ e'}$$

B.1 Standard extensions

Figure 7 lists some type abbreviations used by the CC2-to-linear $F\omega$ translation. This section defines expressions and patterns for these types, based on standard lambda calculus encodings [21]. Some expressions are simple abbreviations, while others use a typing derivation to guide their encoding.

B.1.1 Let bindings

$$\frac{\Delta; \Gamma_1 \vdash e_1 : \phi\tau_1 \rightsquigarrow e_1' \qquad \Delta; \Gamma_2, \phi(x : \tau_1) \vdash e_2 : \tau_2 \rightsquigarrow e_2'}{\Delta; \Gamma_1, \Gamma_2 \vdash (\mathrm{let}\, \phi x = e_1\ \mathrm{in}\, e_2) : \tau_2 \rightsquigarrow (\lambda(\phi x){:}\phi\tau_1.e_2')\ e_1'}$$

B.1.2 Patterns

$$p = _ \mid \phi x \mid \langle p_1, p_2 \rangle \mid [\alpha, p]$$

$\lambda p{:}\tau.e = (\lambda(x){:}\tau.\mathrm{let}\, p = x\ \mathrm{in}\, e)$ where x is fresh
(let $_ = e_1$ in e_2) = (let $!x = e_1$ in e_2) where x is fresh
The sections below define (let $p = e_1$ in e_2) for $p = \phi x$, $p = \langle p_1, p_2 \rangle$, and $p = [\alpha, p]$.

B.1.3 Pairs

$$\frac{\Delta; \phi\Gamma_1 \vdash e_1 : \phi\tau_1 \rightsquigarrow e_1' \qquad \Delta; \phi\Gamma_2 \vdash e_2 : \phi\tau_2 \rightsquigarrow e_2'}{\Delta; \phi\Gamma_1, \phi\Gamma_2 \vdash \phi\langle e_1, e_2 \rangle : \phi(\phi\tau_1 \otimes \phi\tau_2) \rightsquigarrow \phi\lambda\alpha.\lambda f{:}(\phi\tau_1 \multimap \phi\tau_2 \multimap \alpha).f\ e_1'\ e_2'}$$

$$\frac{\Delta; \Gamma_a \vdash e_a : \phi(\phi\tau_1 \otimes \phi\tau_2) \rightsquigarrow e_a' \qquad \Delta; \Gamma_b, x_1 : \phi\tau_1, x_2 : \phi\tau_2 \vdash e_b : \tau_b \rightsquigarrow e_b'}{\Delta; \Gamma_a, \Gamma_b \vdash (\mathrm{letpair}\, x_1, x_2 = e_a\ \mathrm{in}\, e_b) : \tau_b \rightsquigarrow e_a'\ \tau_b\ (\lambda x_1{:}\phi\tau_1.\lambda x_2{:}\phi\tau_2.e_b')}$$

$(\text{let}\,\langle p_1,p_2\rangle = e_a\ \text{in}\, e_b) =$
$\quad(\text{let}\, x = e_a\ \text{in}\,\text{letpair}\, x_1, x_2 = x\,\text{in}\,\text{let}\, p_1 = x_1\ \text{in}\,\text{let}\, p_2 = x_2\ \text{in}\, e_b)$
where x, x_1, x_2 are fresh

$$(e_1, e_2) = !\langle e_1, e_2\rangle$$

$$\#1\, e = (\text{let}\,\langle x_1, _\rangle = e\ \text{in}\, x_1)$$

$$\#2\, e = (\text{let}\,\langle _, x_2\rangle = e\ \text{in}\, x_2)$$

B.1.4 Unit

$$() = !\lambda\alpha.\lambda x\!:\!\alpha.x$$

B.1.5 Existentials

$$\frac{\Delta;\phi\Gamma \vdash e_1 : [\alpha \leftarrow \tau_1](\phi\tau_2) \rightsquigarrow e_1' \qquad \Delta \vdash \tau_1 : \kappa}{\Delta;\phi\Gamma \vdash \phi\text{pack}[\tau_1, e]\,\text{as}\,\phi\exists\alpha\!:\!\kappa.\phi\tau_2 : \phi\exists\alpha\!:\!\kappa.\phi\tau_2 \rightsquigarrow \phi\lambda\beta.\lambda f\!:\!(\forall\alpha\!:\!\kappa.\tau \multimap \beta).f\ \tau_1\ e'}$$

$$\frac{\Delta;\Gamma_a \vdash e_a : \phi\exists\alpha\!:\!\kappa.\phi\tau_2 \rightsquigarrow e_a' \qquad \Delta,\alpha:\kappa;\Gamma_b, x : \phi\tau_2 \vdash e_b : \tau_b \rightsquigarrow e_b' \qquad \Delta \vdash \tau_b : \text{Type}}{\Delta;\Gamma_a,\Gamma_b \vdash (\text{unpack}\,\alpha, x = e_a\ \text{in}\, e_b) : \tau_b \rightsquigarrow e_a'\ \tau_b\ (\lambda\alpha\!:\!\kappa.\lambda x\!:\!\phi\tau_2.e_b')}$$

$(\text{let}\,[\alpha, p] = e_1\ \text{in}\, e_2) = (\text{unpack}\,\alpha, x = e_1\ \text{in}\,\text{let}\, p = x\ \text{in}\, e_2)$ where x is fresh

B.2 Some useful functions

Section C.4's CC2-to-linear $F\omega$ translation relies on the functions below.

$frefl : \tau \rightarrow \tau$
$frefl = !\lambda x\!:\!\tau.x$
$ftrans : (\tau_1 \rightarrow \tau_2) \multimap (\tau_2 \rightarrow \tau_3) \multimap \tau_1 \rightarrow \tau_3$
$ftrans = \lambda\, !x\!:\!\tau_1 \rightarrow \tau_2.\lambda\, !y\!:\!\tau_2 \rightarrow \tau_3.\, !\lambda z\!:\!\tau_1.y\ (x\ z)$
$irefl : \tau \leftrightarrow \tau$
$irefl = !\langle frefl, frefl\rangle$
$isymm : (\tau_2 \leftrightarrow \tau_1) \multimap \tau_1 \leftrightarrow \tau_2$
$isymm = \lambda\langle\, !x_1,\, !x_2\rangle\!:\!\tau_2 \leftrightarrow \tau_1.\, !\langle x_2, x_1\rangle$
$itrans : (\tau_1 \leftrightarrow \tau_2) \multimap (\tau_2 \leftrightarrow \tau_3) \multimap \tau_1 \leftrightarrow \tau_3$
$itrans = \lambda\langle\, !x_1,\, !x_2\rangle\!:\!\tau_1 \leftrightarrow \tau_2.\lambda\langle\, !y_1,\, !y_2\rangle\!:\!\tau_2 \leftrightarrow \tau_3.\, !\langle ftrans\ x_1\ y_1, ftrans\ y_2\ x_2\rangle$
$iexist :\ !(\forall\alpha\!:\!\text{Type}.\, !\tau_1 \leftrightarrow !\tau_2) \multimap !(\exists\alpha\!:\!\text{Type}.\, !\tau_1) \leftrightarrow !(\exists\alpha\!:\!\text{Type}.\, !\tau_2)$
$iexist = \lambda\, !x\!:\!!(\forall\alpha\!:\!\text{Type}.\, !\tau_1 \leftrightarrow !\tau_2).\, !\langle\, !\lambda[\alpha,\, !z]:$
$!(\exists\alpha\!:\!\text{Type}.\, !\tau_1).\, !\text{pack}[\alpha, (\#1\, x\ \alpha)\ z]\,\text{as}\,\exists\alpha\!:\!\text{Type}.\, !\tau_2,\, !\lambda[\alpha,\, !z]:$
$!(\exists\alpha\!:\!\text{Type}.\, !\tau_2).\, !\text{pack}[\alpha, (\#2\, x\ \alpha)\ z]\,\text{as}\,\exists\alpha\!:\!\text{Type}.\, !\tau_1\rangle$
$iall :\ !(\forall\alpha\!:\!\text{Type}.\, !\tau_1 \leftrightarrow !\tau_2) \multimap !(\forall\alpha\!:\!\text{Type}.\, !\tau_1) \leftrightarrow !(\forall\alpha\!:\!\text{Type}.\, !\tau_2)$
$iall = \lambda\, !x\!:\!!(\forall\alpha\!:\!\text{Type}.\, !\tau_1 \leftrightarrow !\tau_2).\, !\langle\, !\lambda\, !y\!:\!!(\forall\alpha\!:\!\text{Type}.\, !\tau_1).\, !\lambda\alpha:$
$\text{Type}.(\#1\, x\ \alpha)\ (y\ \alpha),\, !\lambda\, !y\!:\!!(\forall\alpha\!:\!\text{Type}.\, !\tau_2).\, !\lambda\alpha\!:\!\text{Type}.(\#2\, x\ \alpha)\ (y\ \alpha)\rangle$
$iprod : (\, !\tau_1 \leftrightarrow !\tau_3) \multimap (\, !\tau_2 \leftrightarrow !\tau_4) \multimap !\tau_1 \times !\tau_2 \leftrightarrow !\tau_3 \times !\tau_4$
$iprod = \lambda\langle\, !x_1,\, !x_1'\rangle\!:\!!\tau_1 \leftrightarrow !\tau_3.\lambda\langle\, !x_2,\, !x_2'\rangle\!:\!!\tau_2 \leftrightarrow !\tau_4.\, !\langle\, !\lambda\langle\, !y_1,\, !y_2\rangle:$
$!\tau_1 \times !\tau_2.\, !\langle x_1\ y_1, x_2\ y_2\rangle,\, !\lambda\langle\, !y_1,\, !y_2\rangle\!:\!!\tau_3 \times !\tau_4.\, !\langle x_1'\ y_1, x_2'\ y_2\rangle\rangle$
$iprodleft : (\, !\tau_1 \leftrightarrow !\tau_3) \multimap !\tau_1 \times !\tau_2 \leftrightarrow !\tau_3 \times !\tau_2$
$iprodleft = \lambda x\!:\!!\tau_1 \leftrightarrow !\tau_3.iprod\ x\ irefl$
$ilprod : (\tau_1 \leftrightarrow \tau_3) \multimap (\tau_2 \leftrightarrow \tau_4) \multimap \tau_1 \otimes \tau_2 \leftrightarrow \tau_3 \otimes \tau_4$
$ilprod = \lambda\, !x_1\!:\!\tau_1 \leftrightarrow \tau_3.\lambda\, !x_2\!:\!\tau_2 \leftrightarrow \tau_4.\, !\langle\, !\lambda y\!:\!\tau_1 \otimes \tau_2.\text{let}\,\langle y_1, y_2\rangle = y\ \text{in}$

$\langle (\#1\, x_1)\ y_1, (\#1\, x_2)\ y_2\rangle, !\lambda y\!:\!\tau_3 \otimes \tau_4.\text{let}\ \langle y_1, y_2\rangle = y\ \text{in}$
$\langle (\#2\, x_1)\ y_1, (\#2\, x_2)\ y_2\rangle\rangle$
$\mathit{ilprodright} : (\tau_2 \leftrightarrow \tau_3) \multimap \tau_1 \otimes \tau_2 \leftrightarrow \tau_1 \otimes \tau_3$
$\mathit{ilprodright} = \lambda x\!:\!\tau_2 \leftrightarrow \tau_3.\mathit{ilprod}\ \mathit{irefl}\ x$
$\mathit{ilfun} : (\tau_1 \leftrightarrow \tau_3) \multimap (\tau_2 \leftrightarrow \tau_4) \multimap (\tau_1 \multimap \tau_2) \leftrightarrow \tau_3 \multimap \tau_4$
$\mathit{ilfun} = \lambda\, !x_1\!:\!\tau_1 \leftrightarrow \tau_3.\lambda\, !x_2\!:\!\tau_2 \leftrightarrow \tau_4.\, !\langle\, !\lambda y\!:\!\tau_1 \multimap \tau_2.\lambda z\!:$
$\tau_3.(\#1\, x_2)\ (y\ ((\#2\, x_1)\ z)), !\lambda y\!:\!\tau_3 \multimap \tau_4.\lambda z\!:\!\tau_1.(\#2\, x_2)\ (y\ ((\#1\, x_1)\ z))\rangle$
$\mathit{ilfunleft} : (\tau_1 \leftrightarrow \tau_3) \multimap (\tau_1 \multimap \tau_2) \leftrightarrow \tau_3 \multimap \tau_2$
$\mathit{ilfunleft} = \lambda x\!:\!\tau_1 \leftrightarrow \tau_3.\mathit{ilfun}\ x\ \mathit{irefl}$
$\mathit{ilfunright} : (\tau_2 \leftrightarrow \tau_3) \multimap (\tau_1 \multimap \tau_2) \leftrightarrow \tau_1 \multimap \tau_3$
$\mathit{ilfunright} = \lambda x\!:\!\tau_2 \leftrightarrow \tau_3.\mathit{ilfun}\ \mathit{irefl}\ x$
$\mathit{ifun} : (\tau_1 \leftrightarrow \tau_3) \multimap (\tau_2 \leftrightarrow \tau_4) \multimap (\tau_1 \to \tau_2) \leftrightarrow \tau_3 \to \tau_4$
$\mathit{ifun} = \lambda\, !x_1\!:\!\tau_1 \leftrightarrow \tau_3.\lambda\, !x_2\!:\!\tau_2 \leftrightarrow \tau_4.\, !\langle\, !\lambda\, !y\!:\!\tau_1 \to \tau_2.\, !\lambda z\!:$
$\tau_3.(\#1\, x_2)\ (y\ ((\#2\, x_1)\ z)), !\lambda\, !y\!:\!\tau_3 \to \tau_4.\, !\lambda z\!:\!\tau_1.(\#2\, x_2)\ (y\ ((\#1\, x_1)\ z))\rangle$
$\mathit{ifunright} : (\tau_2 \leftrightarrow \tau_3) \multimap (\tau_1 \to \tau_2) \leftrightarrow \tau_1 \to \tau_3$
$\mathit{ifunright} = \lambda x\!:\!\tau_2 \leftrightarrow \tau_3.\mathit{ifun}\ \mathit{irefl}\ x$
$\mathit{iinter} : (\tau_1 \leftrightarrow \tau_3) \multimap (\tau_2 \leftrightarrow \tau_4) \multimap (\tau_1 \leftrightarrow \tau_2) \leftrightarrow \tau_3 \leftrightarrow \tau_4$
$\mathit{iinter} = \lambda\, !x_1\!:\!\tau_1 \leftrightarrow \tau_3.\lambda\, !x_2\!:\!\tau_2 \leftrightarrow \tau_4.\mathit{iprod}\ (\mathit{ifun}\ x_1\ x_2)\ (\mathit{ifun}\ x_2\ x_1)$
$\mathit{iextractleft} : (\tau_1 \leftrightarrow \tau_3) \multimap (\tau_1 \rightrightarrows \tau_2) \leftrightarrow \tau_3 \rightrightarrows \tau_2$
$\mathit{iextractleft} = \lambda\, !x_1\!:\!\tau_1 \leftrightarrow \tau_3.\mathit{ifun}\ x_1\ (\mathit{ilprodright}\ (\mathit{ilfunright}\ x_1))$
$\mathit{idup} : !\tau \leftrightarrow !\tau \times !\tau$
$\mathit{idup} = !\langle\, !\lambda\, !x\!:\!!\tau.\, !\langle x, x\rangle, !\lambda\, !x\!:\!!\tau \times !\tau.\#1\, x\rangle$
$\mathit{iprodcomm} : !\tau_1 \times !\tau_2 \leftrightarrow !\tau_2 \times !\tau_1$
$\mathit{iprodcomm} = !\langle\, !\lambda\langle\, !x_1, !x_2\rangle\!:\!!\tau_1 \times !\tau_2.\, !\langle x_2, x_1\rangle, !\lambda\langle\, !x_2, !x_1\rangle\!:\!!\tau_2 \times !\tau_1.\, !\langle x_1, x_2\rangle\rangle$
$\mathit{ilprodcomm} : \tau_1 \otimes \tau_2 \leftrightarrow \tau_2 \otimes \tau_1$
$\mathit{ilprodcomm} = !\langle\, !\lambda\langle x_1, x_2\rangle\!:\!\tau_1 \otimes \tau_2.\langle x_2, x_1\rangle, !\lambda\langle x_2, x_1\rangle\!:\!\tau_2 \otimes \tau_1.\langle x_1, x_2\rangle\rangle$
$\mathit{iprodassoc} : !\tau_1 \times (\, !\tau_2 \times !\tau_3) \leftrightarrow !\tau_1 \times !\tau_2 \times !\tau_3$
$\mathit{iprodassoc} = !\langle\, !\lambda\langle\, !x_1, \langle\, !x_2, !x_3\rangle\rangle\!:\!!\tau_1 \times (\, !\tau_2 \times !\tau_3).\, !\langle\, !\langle x_1, x_2\rangle, x_3\rangle, !\lambda\langle\langle\, !x_1, !x_2\rangle, !x_3\rangle\!:$
$!\tau_1 \times !\tau_2 \times !\tau_3.\, !\langle x_1, !\langle x_2, x_3\rangle\rangle\rangle$
$\mathit{ilprodassoc} : \tau_1 \otimes (\tau_2 \otimes \tau_3) \leftrightarrow \tau_1 \otimes \tau_2 \otimes \tau_3$
$\mathit{ilprodassoc} = !\langle\, !\lambda x\!:\!\tau_1 \otimes (\tau_2 \otimes \tau_3).\text{let}\ \langle x_1, y\rangle = x\ \text{in}$
$\text{let}\ \langle x_2, x_3\rangle = y\ \text{in}$
$\langle\langle x_1, x_2\rangle, x_3\rangle, !\lambda x\!:\!\tau_1 \otimes \tau_2 \otimes \tau_3.\text{let}\ \langle y, x_3\rangle = x\ \text{in}$
$\text{let}\ \langle x_1, x_2\rangle = y\ \text{in}$
$\langle x_1, \langle x_2, x_3\rangle\rangle\rangle$
$\mathit{ilprodswapinner} : \tau_1 \otimes \tau_2 \otimes (\tau_3 \otimes \tau_4) \leftrightarrow \tau_1 \otimes \tau_3 \otimes (\tau_2 \otimes \tau_4)$
$\mathit{ilprodswapinner} = !\langle\, !\lambda x\!:\!\tau_1 \otimes \tau_2 \otimes (\tau_3 \otimes \tau_4).\text{let}\ \langle y_1, y_2\rangle = x\ \text{in}$
$\text{let}\ \langle x_1, x_2\rangle = y_1\ \text{in}$
$\text{let}\ \langle x_3, x_4\rangle = y_2\ \text{in}$
$\langle\langle x_1, x_3\rangle, \langle x_2, x_4\rangle\rangle, !\lambda x\!:\!\tau_1 \otimes \tau_3 \otimes (\tau_2 \otimes \tau_4).\text{let}\ \langle y_1, y_2\rangle = x\ \text{in}$
$\text{let}\ \langle x_1, x_2\rangle = y_1\ \text{in}$
$\text{let}\ \langle x_3, x_4\rangle = y_2\ \text{in}$
$\langle\langle x_1, x_3\rangle, \langle x_2, x_4\rangle\rangle\rangle$
$\mathit{iprodunitleftleft} : () \times !\tau \leftrightarrow !\tau$
$\mathit{iprodunitleftleft} = !\langle\, !\lambda\, !x\!:\!() \times !\tau.\#2\, x, !\lambda\, !x\!:\!!\tau.\, !\langle (), x\rangle\rangle$
$\mathit{ilprodunitleftleft} : () \otimes \tau \leftrightarrow \tau$
$\mathit{ilprodunitleftleft} = !\langle\, !\lambda\langle _, x\rangle\!:\!() \otimes \tau.x, !\lambda x\!:\!\tau.\langle (), x\rangle\rangle$
$\mathit{ilprodunitrightleft} : \tau \leftrightarrow () \otimes \tau$
$\mathit{ilprodunitrightleft} = \mathit{isymm}\ \mathit{ilprodunitleftleft}$
$\mathit{ilprodunitrightright} : \tau \leftrightarrow \tau \otimes ()$
$\mathit{ilprodunitrightright} = \mathit{itrans}\ \mathit{ilprodunitrightleft}\ \mathit{ilprodcomm}$
$\mathit{eextractinter} : (\tau_1 \leftrightarrow \tau_2) \multimap \tau_1 \rightrightarrows \tau_2$
$\mathit{eextractinter} = \lambda\langle\, !x, !x'\rangle\!:\!\tau_1 \leftrightarrow \tau_2.\, !\lambda y_1\!:\!\tau_1.\langle x\ y_1, \lambda y_2\!:\!\tau_2.x'\ y_2\rangle$
$\mathit{erefl} : \tau \rightrightarrows \tau$

$erefl = eextractinter\ irefl$
$etrans : (\tau_1 \rightrightarrows \tau_2) \multimap (\tau_2 \rightrightarrows \tau_3) \multimap \tau_1 \rightrightarrows \tau_3$
$etrans = \lambda\,!x_1{:}\tau_1 \rightrightarrows \tau_2.\lambda\,!x_2{:}\tau_2 \rightrightarrows \tau_3.\,!\lambda y_1{:}\tau_1.\text{let}\ \langle y_2, y_2' \rangle = x_1\ y_1\ \text{in}$
$\text{let}\ \langle y_3, y_3' \rangle = x_2\ y_2\ \text{in}$
$\langle y_3, \lambda z_3{:}\tau_3.y_2'\ (y_3'\ z_3)\rangle$
$elprodassoccomm : \tau_2 \otimes \tau_1 \otimes \tau_3 \rightrightarrows \tau_1 \otimes (\tau_2 \otimes \tau_3)$
$elprodassoccomm = !\lambda\langle\langle x_2, x_1\rangle, x_3\rangle{:}\tau_2 \otimes \tau_1 \otimes \tau_3.\langle\langle x_1, \langle x_2, x_3\rangle\rangle, \lambda\langle x_1, \langle x_2, x_3\rangle\rangle :$
$\tau_1 \otimes (\tau_2 \otimes \tau_3).\langle\langle x_2, x_1\rangle, x_3\rangle\rangle$
$egetright : \tau_1 \otimes \tau_2 \rightrightarrows \tau_2$
$egetright = !\lambda\langle x_1, x_2\rangle{:}\tau_1 \otimes \tau_2.\langle x_2, \lambda y_2{:}\tau_2.\langle x_1, y_2\rangle\rangle$
$egetleft : \tau_1 \otimes \tau_2 \rightrightarrows \tau_1$
$egetleft = etrans\ (eextractinter\ ilprodcomm)\ egetright$
$egetleftright : \tau_1 \otimes \tau_2 \otimes \tau_3 \rightrightarrows \tau_2 \otimes \tau_3$
$egetleftright = etrans\ (eextractinter\ (isymm\ ilprodassoc))\ egetright$
$egetleftleft : \tau_1 \otimes \tau_2 \otimes \tau_3 \rightrightarrows \tau_1 \otimes \tau_3$
$egetleftleft = etrans\ (eextractinter\ (ilprod\ ilprodcomm\ irefl))\ egetleftright$
$egetrightleft : \tau_1 \otimes (\tau_2 \otimes \tau_3) \rightrightarrows \tau_1 \otimes \tau_2$
$egetrightleft =$
$etrans\ (etrans\ (eextractinter\ ilprodcomm)\ egetleftleft)\ (eextractinter\ ilprodcomm)$
$egetrightright : \tau_1 \otimes (\tau_2 \otimes \tau_3) \rightrightarrows \tau_1 \otimes \tau_3$
$egetrightright =$
$etrans\ (etrans\ (eextractinter\ ilprodcomm)\ egetleftright)\ (eextractinter\ ilprodcomm)$
$elprod : (\tau_1 \rightrightarrows \tau_3) \multimap (\tau_2 \rightrightarrows \tau_4) \multimap \tau_1 \otimes \tau_2 \rightrightarrows \tau_3 \otimes \tau_4$
$elprod = \lambda\,!x_1{:}\tau_1 \rightrightarrows \tau_3.\lambda\,!x_2{:}\tau_2 \rightrightarrows \tau_4.\,!\lambda\langle y_1, y_2\rangle{:}\tau_1 \otimes \tau_2.\text{let}\ \langle z_1, z_1'\rangle = x_1\ y_1\ \text{in}$
$\text{let}\ \langle z_2, z_2'\rangle = x_2\ y_2\ \text{in}$
$\langle\langle z_1, z_2\rangle, \lambda\langle y_1', y_2'\rangle{:}\tau_3 \otimes \tau_4.\langle z_1'\ y_1', z_2'\ y_2'\rangle\rangle$
$elprodleft : (\tau_1 \rightrightarrows \tau_3) \multimap \tau_1 \otimes \tau_2 \rightrightarrows \tau_3 \otimes \tau_2$
$elprodleft = \lambda x{:}\tau_1 \rightrightarrows \tau_3.elprod\ x\ erefl$
$elprodright : (\tau_2 \rightrightarrows \tau_3) \multimap \tau_1 \otimes \tau_2 \rightrightarrows \tau_1 \otimes \tau_3$
$elprodright = \lambda x{:}\tau_2 \rightrightarrows \tau_3.elprod\ erefl\ x$

C Translations and lemmas

Sections C.1, C.2, C.3, and C.4 define the CC0-to-CC1, CC1-to-CC2, CC2-to-LC, and CC2-to-linear $F\omega$ translations. They also state the lemmas that constitute the proof of the translation type correctness. The lemmas are proved in [12].

C.1 Translation: CC0→CC1

$\mathcal{C}(\alpha) = \alpha$
$\mathcal{C}(\emptyset) = \emptyset$
$\mathcal{C}(\{\alpha^\varphi\}) = \{\alpha^\varphi\}$
$\mathcal{C}(C_1 \oplus C_2) = \mathcal{C}(C_1) \oplus \mathcal{C}(C_2)$
$\mathcal{C}(\overline{C}) = \mathcal{S}(C)$
$\mathcal{S}(\alpha) = \alpha_S$
$\mathcal{S}(\emptyset) = \emptyset$
$\mathcal{S}(\{\alpha^\varphi\}) = \{\alpha^+\}$
$\mathcal{S}(C_1 \oplus C_2) = \mathcal{S}(C_1) \oplus \mathcal{S}(C_2)$
$\mathcal{S}(\overline{C}) = \mathcal{S}(C)$
$\mathcal{T}(\alpha) = \alpha$
$\mathcal{T}(\rho\,\text{handle}) = \rho\,\text{handle}$
$\mathcal{T}((C, \tau) \to 0) = (\mathcal{C}(C), \mathcal{T}(\tau)) \to 0$

$\mathcal{T}(\tau_1 \times \tau_2) = \mathcal{T}(\tau_1) \times \mathcal{T}(\tau_2)$
$\mathcal{T}(\forall\alpha{:}\mathrm{Type}.\tau) = \forall\alpha{:}\mathrm{Type}.\mathcal{T}(\tau)$
$\mathcal{T}(\forall\rho{:}\mathrm{Res}.\tau) = \forall\rho{:}\mathrm{Res}.\mathcal{T}(\tau)$
$\mathcal{T}(\forall\alpha{:}\mathrm{Cap}.\tau) = \forall\alpha_S{:}\mathrm{Cap}^+.\forall\alpha{:}\mathrm{Cap} \leq \alpha_S.\mathcal{T}(\tau)$
$\mathcal{T}(\forall\alpha \leq C.\tau) = \forall\alpha_S{:}\mathrm{Cap}^+ \leq \mathcal{S}(C).\forall\alpha{:}\mathrm{Cap} \leq \mathcal{C}(C), \alpha_S.\mathcal{T}(\tau)$
$\mathbf{\Delta}(\cdot) = \cdot$
$\mathbf{\Delta}(\alpha : \mathrm{Type}, \Delta) = \alpha : \mathrm{Type}, \mathbf{\Delta}(\Delta)$
$\mathbf{\Delta}(\alpha : \mathrm{Res}, \Delta) = \alpha : \mathrm{Res}, \mathbf{\Delta}(\Delta)$
$\mathbf{\Delta}(\alpha : \mathrm{Cap}, \Delta) = \alpha_S : \mathrm{Cap}^+, \alpha : \mathrm{Cap} \leq \alpha_S, \mathbf{\Delta}(\Delta)$
$\mathbf{\Delta}(\alpha \leq C, \Delta) = \alpha_S : \mathrm{Cap}^+ \leq \mathcal{S}(C), \alpha : \mathrm{Cap} \leq (\mathcal{C}(C), \alpha_S), \mathbf{\Delta}(\Delta)$
$\mathbf{\Gamma}(\cdot) = \cdot$
$\mathbf{\Gamma}(x : \tau, \Gamma) = x : \mathcal{T}(\tau), \mathbf{\Gamma}(\Gamma)$
$\mathcal{V}(x) = x$
$\mathcal{V}(v[\tau : \mathrm{Type}]) = \mathcal{V}(v)[\mathcal{T}(\tau) : \mathrm{Type}]$
$\mathcal{V}(v[\alpha : \mathrm{Res}]) = \mathcal{V}(v)[\alpha : \mathrm{Res}]$
$\mathcal{V}(v[C : \mathrm{Cap}]) = \mathcal{V}(v)[\mathcal{S}(C) : \mathrm{Cap}^+][\mathcal{C}(C) : \mathrm{Cap}]$
$\mathcal{H}(\lambda\alpha{:}\mathrm{Type}.h) = \lambda\alpha{:}\mathrm{Type}.\mathcal{H}(h)$
$\mathcal{H}(\lambda\alpha{:}\mathrm{Res}.h) = \lambda\alpha{:}\mathrm{Res}.\mathcal{H}(h)$
$\mathcal{H}(\lambda\alpha{:}\mathrm{Cap}.h) = \lambda\alpha_S{:}\mathrm{Cap}^+.\lambda\alpha{:}\mathrm{Cap} \leq \alpha_S.\mathcal{H}(h)$
$\mathcal{H}(\lambda\alpha \leq C.h) = \lambda\alpha_S{:}\mathrm{Cap}^+ \leq \mathcal{S}(C).\lambda\alpha{:}\mathrm{Cap} \leq \mathcal{C}(C), \alpha_S.\mathcal{H}(h)$
$\mathcal{H}(\lambda(C, x : \tau).e) = \lambda(\mathcal{C}(C), x : \mathcal{T}(\tau)).\mathcal{E}(e)$
$\mathcal{H}((v_1, v_2)) = (\mathcal{V}(v_1), \mathcal{V}(v_2))$
$\mathcal{D}(x = v) = (x = \mathcal{V}(v))$
$\mathcal{D}(x = h) = (x = \mathcal{H}(h))$
$\mathcal{D}(x = \#n\, v) = (x = \#n\, \mathcal{V}(v))$
$\mathcal{D}(\mathrm{new}\,\alpha, x) = \mathrm{new}\,\alpha, x$
$\mathcal{D}(\mathrm{free}\, v) = \mathrm{free}\,\mathcal{V}(v)$
$\mathcal{D}(\mathrm{use}\, v) = \mathrm{use}\,\mathcal{V}(v)$
$\mathcal{E}(\mathrm{let}\, d \,\mathrm{in}\, e) = \mathrm{let}\,\mathcal{D}(d) \,\mathrm{in}\, \mathcal{E}(e)$
$\mathcal{E}(v_1\ v_2) = \mathcal{V}(v_1)\ \mathcal{V}(v_2)$
$\mathcal{E}(\mathrm{halt}) = \mathrm{halt}$

C.1.1 Lemmas for CC0→CC1

- If $\Delta \vdash C : \mathrm{Cap}$ then $\mathbf{\Delta}(\Delta) \vdash \mathcal{S}(C) : \mathrm{Cap}^+$
- If $\Delta \vdash C : \mathrm{Cap}$ then $\mathbf{\Delta}(\Delta) \vdash \mathcal{C}(C) : \mathrm{Cap}$
- If $\Delta \vdash \tau : \mathrm{Type}$ then $\mathbf{\Delta}(\Delta) \vdash \mathcal{T}(\tau) : \mathrm{Type}$
- If $\Delta \vdash \Delta'$ then $\mathbf{\Delta}(\Delta) \vdash \mathbf{\Delta}(\Delta')$
- If $\Delta \vdash C_1 = C_2 : \mathrm{Cap}$ then $\mathbf{\Delta}(\Delta) \vdash \mathcal{S}(C_1) = \mathcal{S}(C_2) : \mathrm{Cap}^+$
- If $\Delta \vdash C_1 = C_2 : \mathrm{Cap}$ then $\mathbf{\Delta}(\Delta) \vdash \mathcal{C}(C_1) = \mathcal{C}(C_2) : \mathrm{Cap}$
- If $\Delta \vdash \tau_1 = \tau_2 : \mathrm{Type}$ then $\mathbf{\Delta}(\Delta) \vdash \mathcal{T}(\tau_1) = \mathcal{T}(\tau_2) : \mathrm{Type}$
- If $\Delta \vdash C : \mathrm{Cap}$ then $\mathbf{\Delta}(\Delta) \vdash \mathcal{C}(C) \leq \mathcal{S}(C)$
- If $\Delta \vdash C_1 \leq C_2$ then $\mathbf{\Delta}(\Delta) \vdash \mathcal{S}(C_1) \leq \mathcal{S}(C_1)$
- If $\Delta \vdash C_1 \leq C_2$ then $\mathbf{\Delta}(\Delta) \vdash \mathcal{C}(C_1) \leq \mathcal{C}(C_2)$
- If $\vdash \Delta$ and $\Delta \vdash \alpha' : \mathrm{Cap}$ and $\Delta \vdash C : \mathrm{Cap}$ then $\mathcal{S}([\alpha' \leftarrow C']C) = [\alpha' \leftarrow \mathcal{C}(C'), \alpha'_S \leftarrow \mathcal{S}(C')]\mathcal{S}(C)$

- If $\vdash \Delta$ and $\Delta \vdash \alpha' : \text{Cap}$ and $\Delta \vdash C : \text{Cap}$ then $\mathcal{C}([\alpha' \leftarrow C']C) = [\alpha' \leftarrow \mathcal{C}(C'), \alpha'_S \leftarrow \mathcal{S}(C')]\mathcal{C}(C)$
- If $\vdash \Delta$ and $\Delta \vdash \alpha' : \text{Cap}$ and $\Delta \vdash \tau : \text{Type}$ then $\mathcal{T}([\alpha' \leftarrow C']\tau) = [\alpha' \leftarrow \mathcal{C}(C'), \alpha'_S \leftarrow \mathcal{S}(C')]\mathcal{T}(\tau)$
- If $\vdash \Delta$ and $\Delta \vdash \alpha' : \text{Type}$ and $\Delta \vdash \tau : \text{Type}$ then $\mathcal{T}([\alpha' \leftarrow \tau']\tau) = [\alpha' \leftarrow \mathcal{T}(\tau')]\mathcal{T}(\tau)$
- If $\vdash \Delta$ and $\Delta \vdash \Gamma$ then $\mathbf{\Delta}(\Delta) \vdash \mathbf{\Gamma}(\Gamma)$
- If $\vdash \Delta$ and $\Delta \vdash \Gamma$ then:
  - If $\Delta; \Gamma \vdash v : \tau$ then $\mathbf{\Delta}(\Delta); \mathbf{\Gamma}(\Gamma) \vdash \mathcal{V}(v) : \mathcal{T}(\tau)$
  - If $\Delta; \Gamma \vdash h : \tau$ then $\mathbf{\Delta}(\Delta); \mathbf{\Gamma}(\Gamma) \vdash \mathcal{H}(h) : \mathcal{T}(\tau)$
  - If $\Delta; \Gamma; C \vdash d \Longrightarrow \Delta'; \Gamma'; C'$ then $\mathbf{\Delta}(\Delta); \mathbf{\Gamma}(\Gamma); \mathcal{C}(C) \vdash \mathcal{D}(d) \Longrightarrow \mathbf{\Delta}(\Delta'); \mathbf{\Gamma}(\Gamma'); \mathcal{C}(C')$
  - If $\Delta; \Gamma; C \vdash e$ then $\mathbf{\Delta}(\Delta); \mathbf{\Gamma}(\Gamma); \mathcal{C}(C) \vdash \mathcal{E}(e)$

C.2 Translation: CC1→CC2

$$
\begin{array}{l}
\mathcal{C}(C) = \mathcal{U}(C) \boxplus \mathcal{A}(C) \\
\mathcal{U}(\alpha) = \alpha_U \\
\mathcal{U}(\emptyset) = \emptyset \\
\mathcal{U}(\{\rho^1\}) = \{\rho^1\} \\
\mathcal{U}(\{\rho^+\}) = \emptyset \\
\mathcal{U}(C_1 \oplus C_2) = \mathcal{U}(C_1) \oplus \mathcal{U}(C_2) \\
\mathcal{A}(\alpha) = \alpha_A \\
\mathcal{A}(\emptyset) = \emptyset \\
\mathcal{A}(\{\rho^1\}) = \emptyset \\
\mathcal{A}(\{\rho^+\}) = \{\rho^+\} \\
\mathcal{A}(C_1 \oplus C_2) = \mathcal{A}(C_1) \oplus \mathcal{A}(C_2) \\
\mathcal{T}(\alpha) = \alpha \\
\mathcal{T}(\rho\,\text{handle}) = \rho\,\text{handle} \\
\mathcal{T}((C, \tau) \to 0) = (\mathcal{C}(C), \mathcal{T}(\tau)) \to 0 \\
\mathcal{T}(\tau_1 \times \tau_2) = \mathcal{T}(\tau_1) \times \mathcal{T}(\tau_2) \\
\mathcal{T}(\forall \alpha{:}\text{Type}.\tau) = \forall \alpha{:}\text{Type}.\mathcal{T}(\tau) \\
\mathcal{T}(\forall \alpha{:}\text{Res}.\tau) = \forall \alpha{:}\text{Res}.\mathcal{T}(\tau) \\
\mathcal{T}(\forall \alpha{:}\text{Cap}.\tau) = \forall \alpha_A{:}\text{Cap}^+.\forall \alpha_U{:}\text{Cap}^1.\mathcal{T}(\tau) \\
\mathcal{T}(\forall \alpha{:}\text{Cap}^+.\tau) = \forall \alpha_A{:}\text{Cap}^+.[\alpha_U \leftarrow \emptyset]\mathcal{T}(\tau) \\
\mathcal{T}(\forall \alpha{:}\text{Cap}^+ \le C.\tau) = \forall \alpha_A{:}\text{Cap}^+ \text{ where } \emptyset \boxplus \alpha_A \le \mathcal{A}(C).[\alpha_U \leftarrow \emptyset]\mathcal{T}(\tau) \\
\mathcal{T}(\forall \alpha{:}\text{Cap} \le C_0, C_1, \ldots, C_n.\tau) = \\
\qquad \forall \alpha_B{:}\text{Cap}^1.\forall \alpha_A{:}\text{Cap}^+ \text{ where } \begin{array}[t]{r} \alpha_B \boxplus \alpha_A \le \mathcal{A}(C_0), \\ (\alpha_B \oplus \mathcal{U}(C_0)) \boxplus \alpha_A \le \mathcal{A}(C_1), \\ \vdots \\ (\alpha_B \oplus \mathcal{U}(C_0)) \boxplus \alpha_A \le \mathcal{A}(C_n).[\alpha_U \leftarrow (\alpha_B \oplus \mathcal{U}(C_0))]\mathcal{T}(\tau) \end{array} \\
\mathbf{\Delta}(\cdot) = \cdot \\
\mathbf{\Delta}(\alpha : \text{Type}, \Delta) = \alpha : \text{Type}, \mathbf{\Delta}(\Delta) \\
\mathbf{\Delta}(\alpha : \text{Res}, \Delta) = \alpha : \text{Res}, \mathbf{\Delta}(\Delta) \\
\mathbf{\Delta}(\alpha : \text{Cap}, \Delta) = \alpha_A : \text{Cap}^+, \alpha_U : \text{Cap}^1, \mathbf{\Delta}(\Delta) \\
\mathbf{\Delta}(\alpha : \text{Cap}^+, \Delta) = \alpha_A : \text{Cap}^+, [\alpha_U \leftarrow \emptyset]\mathbf{\Delta}(\Delta) \\
\mathbf{\Delta}(\alpha : \text{Cap}^+ \le C, \Delta) = \alpha_A{:}\text{Cap}^+ \text{ where } \emptyset \boxplus \alpha_A \le \mathcal{A}(C), [\alpha_U \leftarrow \emptyset]\mathbf{\Delta}(\Delta)
\end{array}
$$

$$\mathbf{\Delta}(\alpha : \text{Cap} \le (C_0, C_1, \ldots, C_n), \Delta) = \\ \alpha_B : \text{Cap}^1, \alpha_A{:}\text{Cap}^+ \text{ where } \quad \alpha_B \boxplus \alpha_A \le \mathcal{A}(C_0), \\ (\alpha_B \oplus \mathcal{U}(C_0)) \boxplus \alpha_A \le \mathcal{A}(C_1), \\ \vdots \\ (\alpha_B \oplus \mathcal{U}(C_0)) \boxplus \alpha_A \le \mathcal{A}(C_n), [\alpha_U \leftarrow (\alpha_B \oplus \mathcal{U}(C_0))]\mathbf{\Delta}(\Delta)$$

$[\![\cdot]\!] = []$
$[\![\alpha : \text{Type}, \Delta]\!] = [\![\Delta]\!]$
$[\![\alpha : \text{Res}, \Delta]\!] = [\![\Delta]\!]$
$[\![\alpha : \text{Cap}, \Delta]\!] = [\![\Delta]\!]$
$[\![\alpha : \text{Cap}^+, \Delta]\!] = [\alpha_U \leftarrow \emptyset][\![\Delta]\!]$
$[\![\alpha : \text{Cap}^+ \le C, \Delta]\!] = [\alpha_U \leftarrow \emptyset][\![\Delta]\!]$
$[\![\alpha : \text{Cap} \le (C_0, C_1, \ldots, C_n), \Delta]\!] = [\alpha_U \leftarrow \alpha_B \oplus \mathcal{U}(C_0)][\![\Delta]\!]$
$\mathbf{\Gamma}(\cdot) = \cdot$
$\mathbf{\Gamma}(x : \tau, \Gamma) = x : \mathcal{T}(\tau), \mathbf{\Gamma}(\Gamma)$
The translations of v, h, d, and e are directed by typing judgments:

- $\Delta; \Gamma \vdash v : \tau \rightsquigarrow \mathcal{V}(v)$
- $\Delta; \Gamma \vdash h : \tau \rightsquigarrow \mathcal{H}(h)$
- $\Delta; \Gamma; C \vdash d \Longrightarrow \Delta'; \Gamma'; C' \rightsquigarrow \mathcal{D}(d)$
- $\Delta; \Gamma; C \vdash e \rightsquigarrow \mathcal{E}(e)$

For conciseness, though, most of the definitions below suppress the typing judgment when it is not immediately relevant.
$\mathcal{V}(x) = x$
$\mathcal{V}(v[\tau : \text{Type}]) = \mathcal{V}(v)[\mathcal{T}(\tau) : \text{Type}]$
$\mathcal{V}(v[\alpha : \text{Res}]) = \mathcal{V}(v)[\alpha : \text{Res}]$

$$\frac{\Delta; \Gamma \vdash v : \forall\alpha{:}\kappa.\tau \rightsquigarrow v' \qquad \Delta \vdash c : \kappa}{\Delta; \Gamma \vdash v[c : \kappa] : [\alpha \leftarrow c]\tau \rightsquigarrow v'[\mathcal{A}(C) : \text{Cap}^+][\mathcal{U}(C) : \text{Cap}^1]} (\kappa = \text{Cap})$$

$$\frac{\Delta; \Gamma \vdash v : \forall\alpha{:}\kappa.\tau \rightsquigarrow v' \qquad \Delta \vdash c : \kappa}{\Delta; \Gamma \vdash v[c : \kappa] : [\alpha \leftarrow c]\tau \rightsquigarrow v'[\mathcal{A}(C) : \text{Cap}^+]} (\kappa = \text{Cap}^+)$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash v : \forall\alpha{:}\kappa \le C_0.\tau \rightsquigarrow v' \\ \Delta \vdash C \le C_0 \\ \Delta \vdash C : \kappa\end{array}}{\Delta; \Gamma \vdash v[C : \kappa] : [\alpha \leftarrow C]\tau \rightsquigarrow v'[\mathcal{A}(C) : \text{Cap}^+]} (\kappa = \text{Cap}^+)$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash v : \forall\alpha{:}\kappa \le C_0, C_1, \ldots, C_n.\tau \rightsquigarrow v' \\ \Delta \vdash C \le C_0 \rightsquigarrow U_B \\ \Delta \vdash C \le C_1 \qquad \Delta \vdash C \le C_n \\ \Delta \vdash C : \kappa\end{array}}{\Delta; \Gamma \vdash v[C : \kappa] : [\alpha \leftarrow C]\tau \rightsquigarrow v'[U_B : \text{Cap}^1][\mathcal{A}(C) : \text{Cap}^+]} (\kappa = \text{Cap})$$

$$\frac{\Delta; \Gamma \vdash v : \tau' \rightsquigarrow v' \qquad \Delta \vdash \tau' = \tau : \text{Type}}{\Delta; \Gamma \vdash v : \tau \rightsquigarrow v'}$$

$\mathcal{H}(\lambda\alpha{:}\text{Type}.h) = \lambda\alpha{:}\text{Type}.\mathcal{H}(h)$
$\mathcal{H}(\lambda\alpha{:}\text{Res}.h) = \lambda\alpha{:}\text{Res}.\mathcal{H}(h)$

$\mathcal{H}(\lambda\alpha\!:\!\mathrm{Cap}.h) = \lambda\alpha_A\!:\!\mathrm{Cap}^+.\lambda\alpha_U\!:\!\mathrm{Cap}^1.\mathcal{H}(h)$
$\mathcal{H}(\lambda\alpha\!:\!\mathrm{Cap}^+.h) = \lambda\alpha_A\!:\!\mathrm{Cap}^+.[\alpha_U \leftarrow \emptyset]\mathcal{H}(h)$
$\mathcal{H}(\lambda\alpha\!:\!\mathrm{Cap}^+ \leq C.h) = \lambda\alpha_A\!:\!\mathrm{Cap}^+ \text{ where } \emptyset \boxplus \alpha_A \leq \mathcal{A}(C).[\alpha_U \leftarrow \emptyset]\mathcal{H}(h)$

$$\begin{aligned}
&\mathcal{H}(\lambda\alpha\!:\!\mathrm{Cap} \leq C_0, C_1, \ldots, C_n.h) = \\
&\qquad \lambda\alpha_B\!:\!\mathrm{Cap}^1.\lambda\alpha_A\!:\!\mathrm{Cap}^+ \text{ where } \quad \alpha_B \boxplus \alpha_A \leq \mathcal{A}(C_0), \\
&\qquad\qquad (\alpha_B \oplus \mathcal{U}(C_0)) \boxplus \alpha_A \leq \mathcal{A}(C_1), \\
&\qquad\qquad \vdots \\
&\qquad\qquad (\alpha_B \oplus \mathcal{U}(C_0)) \boxplus \alpha_A \leq \mathcal{A}(C_n).[\alpha_U \leftarrow (\alpha_B \oplus \mathcal{U}(C_0))]\mathcal{H}(h)
\end{aligned}$$

$\mathcal{H}(\lambda(C, x : \tau).e) = \lambda(\mathcal{C}(C), x : \mathcal{T}(\tau)).\mathcal{E}(e)$
$\mathcal{H}((v_1, v_2)) = (\mathcal{V}(v_1), \mathcal{V}(v_2))$

$$\frac{\Delta; \Gamma \vdash h : \tau' \rightsquigarrow h' \qquad \Delta \vdash \tau' = \tau : \mathrm{Type}}{\Delta; \Gamma \vdash h : \tau \rightsquigarrow h'}$$

$\mathcal{D}(x = v) = (x = \mathcal{V}(v))$
$\mathcal{D}(x = h) = (x = \mathcal{H}(h))$
$\mathcal{D}(x = \#n\, v) = (x = \#n\, \mathcal{V}(v))$
$\mathcal{D}(\mathrm{new}\, \alpha, x) = \mathrm{new}\, \alpha, x$
$\mathcal{D}(\mathrm{free}\, v) = \mathrm{free}\, \mathcal{V}(v)$
$\mathcal{D}(\mathrm{use}\, v) = \mathrm{use}\, \mathcal{V}(v)$
$\mathcal{E}(\mathrm{let}\, d \text{ in } e) = \mathrm{let}\, \mathcal{D}(d) \text{ in } \mathcal{E}(e)$
$\mathcal{E}(v_1\ v_2) = \mathcal{V}(v_1)\ \mathcal{V}(v_2)$
$\mathcal{E}(\mathrm{halt}) = \mathrm{halt}$

$$\frac{\Delta \vdash C_1 = C_2 : \kappa}{\Delta \vdash C_1 \leq C_2 \rightsquigarrow \emptyset}(\kappa \in \{\mathrm{Cap}, \mathrm{Cap}^+\})$$

$$\frac{\Delta \vdash C_1 \leq C_2 \rightsquigarrow U \qquad \Delta \vdash C_2 \leq C_3 \rightsquigarrow U'}{\Delta \vdash C_1 \leq C_3 \rightsquigarrow U \oplus U'}$$

$$\frac{\Delta \vdash C_1 \leq C_1' \rightsquigarrow U_1 \qquad \Delta \vdash C_2 \leq C_2' \rightsquigarrow U_2}{\Delta \vdash C_1 \oplus C_2 \leq C_1' \oplus C_2' \rightsquigarrow U_1 \oplus U_2}$$

$$\ldots, \alpha : \mathrm{Cap}^+ \leq C, \ldots \vdash \alpha \leq C \rightsquigarrow \emptyset$$

$$\ldots, \alpha : \mathrm{Cap} \leq (C_0, \ldots), \ldots \vdash \alpha \leq C_0 \rightsquigarrow \alpha_B$$

$$\ldots, \alpha : \mathrm{Cap} \leq (C_0, \ldots, C_k, \ldots), \ldots \vdash \alpha \leq C_k \rightsquigarrow \alpha_B \oplus \mathcal{U}(C_0)$$

$$\frac{\Delta \vdash \alpha : \mathrm{Res}}{\Delta \vdash \{\alpha^1\} \leq \{\alpha^+\} \rightsquigarrow \{\alpha^1\}}$$

C.2.1 Lemmas for CC1→CC2

- If $\vdash \Delta$ and $\Delta \vdash C : \kappa$ and $\kappa \in \{\mathrm{Cap}, \mathrm{Cap}^+\}$ then $\mathbf{\Delta}(\Delta) \vdash [\Delta]\mathcal{U}(C) : \mathrm{Cap}^1$
- If $\vdash \Delta$ and $\Delta \vdash C : \kappa$ and $\kappa \in \{\mathrm{Cap}, \mathrm{Cap}^+\}$ then $\mathbf{\Delta}(\Delta) \vdash [\Delta]\mathcal{A}(C) : \mathrm{Cap}^+$
- If $\vdash \Delta$ and $\Delta \vdash C_1 \leq C_2 \rightsquigarrow U$ then $\mathbf{\Delta}(\Delta) \vdash [\Delta]U : \mathrm{Cap}^1$
- If $\Delta \vdash \Delta'$ then $\mathbf{\Delta}(\Delta) \vdash [\Delta]\mathbf{\Delta}(\Delta')$

- If $\vdash \Delta$ and $\Delta \vdash \tau : \text{Type}$ then $\mathbf{\Delta}(\Delta) \vdash [\Delta]\mathcal{T}(\tau) : \text{Type}$
- If $\vdash \Delta$ and $\Delta \vdash C : \text{Cap}^+$ then $\mathbf{\Delta}(\Delta) \vdash [\Delta]\mathcal{U}(C) = \emptyset : \text{Cap}^1$
- If $\vdash \Delta$ and $\Delta \vdash C_1 = C_2 : \kappa$ and $\kappa \in \{\text{Cap}, \text{Cap}^+\}$ then $\mathbf{\Delta}(\Delta) \vdash [\Delta]\mathcal{U}(C_1) = [\Delta]\mathcal{U}(C_2) : \text{Cap}^1$
- If $\vdash \Delta$ and $\Delta \vdash C_1 = C_2 : \kappa$ and $\kappa \in \{\text{Cap}, \text{Cap}^+\}$ then $\mathbf{\Delta}(\Delta) \vdash [\Delta]\mathcal{A}(C_1) = [\Delta]\mathcal{A}(C_2) : \text{Cap}^+$
- If $\vdash \Delta$ and $\Delta \vdash \tau_1 = \tau_2 : \text{Type}$ then $\mathbf{\Delta}(\Delta) \vdash [\Delta]\mathcal{T}(\tau_1) = [\Delta]\mathcal{T}(\tau_2) : \text{Type}$
- If $\vdash \Delta$ and $\Delta \vdash C_1 \leq C_2 \rightsquigarrow U$ then $\mathbf{\Delta}(\Delta) \vdash [\Delta]\mathcal{U}(C_1) = [\Delta]U \oplus [\Delta]\mathcal{U}(C_2) : \text{Cap}^1$ and $\mathbf{\Delta}(\Delta) \vdash [\Delta]U \boxplus [\Delta]\mathcal{A}(C_1) \leq [\Delta]\mathcal{A}(C_2)$
- If $\vdash \Delta$ and $\Delta \vdash \alpha' : \kappa'$ and $\kappa' \in \{\text{Cap}, \text{Cap}^+\}$ and $\Delta \vdash C : \kappa$ and $\kappa \in \{\text{Cap}, \text{Cap}^+\}$ then $\mathcal{A}([\alpha' \leftarrow C']C) = [\alpha'_U \leftarrow \mathcal{U}(C'), \alpha'_A \leftarrow \mathcal{A}(C')]\mathcal{A}(C)$
- If $\vdash \Delta$ and $\Delta \vdash \alpha' : \kappa'$ and $\kappa' \in \{\text{Cap}, \text{Cap}^+\}$ and $\Delta \vdash C : \kappa$ and $\kappa \in \{\text{Cap}, \text{Cap}^+\}$ then $\mathcal{U}([\alpha' \leftarrow C']C) = [\alpha'_U \leftarrow \mathcal{U}(C'), \alpha'_A \leftarrow \mathcal{A}(C')]\mathcal{U}(C)$
- If $\vdash \Delta$ and $\Delta \vdash \alpha' : \kappa'$ and $\kappa' \in \{\text{Cap}, \text{Cap}^+\}$ and $\Delta \vdash \tau : \text{Type}$ then $\mathcal{T}([\alpha' \leftarrow C']\tau) = [\alpha'_U \leftarrow \mathcal{U}(C'), \alpha'_A \leftarrow \mathcal{A}(C')]\mathcal{T}(\tau)$
- If $\vdash \Delta$ and $\Delta \vdash \alpha' : \text{Type}$ and $\Delta \vdash \tau : \text{Type}$ then $\mathcal{T}([\alpha' \leftarrow \tau']\tau) = [\alpha' \leftarrow \mathcal{T}(\tau')]\mathcal{T}(\tau)$
- If $\vdash \Delta$ and $\Delta \vdash \Gamma$ then $\mathbf{\Delta}(\Delta) \vdash [\Delta]\mathbf{\Gamma}(\Gamma)$
- If $\vdash \Delta$ and $\Delta \vdash \Gamma$ then:
  - If $\Delta; \Gamma \vdash v : \tau \rightsquigarrow v'$ then $\mathbf{\Delta}(\Delta); [\Delta]\mathbf{\Gamma}(\Gamma) \vdash [\Delta]v' : [\Delta]\mathcal{T}(\tau)$
  - If $\Delta; \Gamma \vdash h : \tau \rightsquigarrow h'$ then $\mathbf{\Delta}(\Delta); [\Delta]\mathbf{\Gamma}(\Gamma) \vdash [\Delta]h' : [\Delta]\mathcal{T}(\tau)$
  - If $\Delta; \Gamma; C \vdash d \Longrightarrow \Delta'; \Gamma'; C' \rightsquigarrow d'$ then $\mathbf{\Delta}(\Delta); [\Delta]\mathbf{\Gamma}(\Gamma); [\Delta]\mathcal{C}(C) \vdash [\Delta]d' \Longrightarrow \mathbf{\Delta}(\Delta'); [\Delta]\mathbf{\Gamma}(\Gamma'); [\Delta]\mathcal{C}(C')$ and $[\Delta] = [\Delta']$
  - If $\Delta; \Gamma; C \vdash e \rightsquigarrow e'$ then $\mathbf{\Delta}(\Delta); [\Delta]\mathbf{\Gamma}(\Gamma); [\Delta]\mathcal{C}(C) \vdash [\Delta]e'$

C.3 Translation: CC2→LC

$$
\begin{array}{l}
\mathcal{K}(\text{Type}) = \text{Type} \\
\mathcal{K}(\text{Res}) = \text{Res} \\
\mathcal{K}(\text{Cap}^{\varphi}) = \text{Cap} \\
\mathcal{T}(\alpha) = \alpha \\
\mathcal{T}(\rho\,\text{handle}) = \rho\,\text{handle} \\
\mathcal{T}((C,\tau) \rightarrow 0) = (\mathcal{C}(C), \mathcal{T}(\tau)) \rightarrow 0 \\
\mathcal{T}(\tau_1 \times \tau_2) = \mathcal{T}(\tau_1) \times \mathcal{T}(\tau_2) \\
\mathcal{T}(\forall \alpha{:}\kappa.\tau) = \forall \alpha{:}\mathcal{K}(\kappa).\mathcal{T}(\tau) \\
\mathcal{T}(\forall \alpha{:}\text{Cap}^+\,\text{where}\,U_1 \boxplus \alpha \leq A_1, \ldots, U_n \boxplus \alpha \leq A_n.\tau) = \\
\quad \forall \alpha{:}\text{Cap}.[\alpha \leftarrow (\alpha \& (\mathcal{U}(U_1) \multimap \mathcal{A}(A_1) \otimes \text{true}) \& \ldots \& (\mathcal{U}(U_n) \multimap \mathcal{A}(A_n) \otimes \text{true}))]\mathcal{T}(\tau) \\
\mathcal{C}(U \boxplus A) = \mathcal{U}(U) \otimes (\mathcal{A}(A) \otimes \text{true}) \\
\mathcal{U}(\alpha) = \alpha \\
\mathcal{U}(\emptyset) = \emptyset \\
\mathcal{U}(\{\rho^{\varphi}\}) = \{\rho\} \\
\mathcal{U}(U_1 \oplus U_2) = \mathcal{U}(U_1) \otimes \mathcal{U}(U_2) \\
\mathcal{A}(\alpha) = \alpha \\
\mathcal{A}(\emptyset) = \emptyset \\
\mathcal{A}(\{\rho^{\varphi}\}) = \{\rho\}
\end{array}
$$

$\mathcal{A}(A_1 \oplus A_2) = (\mathcal{A}(A_1) \otimes \text{true})\&(\mathcal{A}(A_2) \otimes \text{true})$
$\mathbf{\Delta}(\cdot) = \cdot$
$\mathbf{\Delta}(\alpha : \kappa, \Delta) = \alpha : \mathcal{K}(\kappa), \mathbf{\Delta}(\Delta)$
$\mathbf{\Delta}(\alpha{:}\text{Cap}^+ \text{ where } U_1 \boxplus \alpha \leq A_1, \ldots, U_n \boxplus \alpha \leq A_n, \Delta) = \alpha : \text{Cap},$
$\quad [\alpha \leftarrow (\alpha\&(\mathcal{U}(U_1) \multimap \mathcal{A}(A_1) \otimes \text{true})\& \ldots \&(\mathcal{U}(U_n) \multimap \mathcal{A}(A_n) \otimes \text{true}))]\mathbf{\Delta}(\Delta)$
$[\![\cdot]\!] = []$
$[\![\alpha : \kappa, \Delta]\!] = [\![\Delta]\!]$
$[\![\alpha{:}\text{Cap}^+ \text{ where } U_1 \boxplus \alpha \leq A_1, \ldots, U_n \boxplus \alpha \leq A_n, \Delta]\!] =$
$\quad [\alpha \leftarrow (\alpha\&(\mathcal{U}(U_1) \multimap \mathcal{A}(A_1) \otimes \text{true})\& \ldots \&(\mathcal{U}(U_n) \multimap \mathcal{A}(A_n) \otimes \text{true}))][\![\Delta]\!]$
$\mathbf{\Gamma}(\cdot) = \cdot$
$\mathbf{\Gamma}(x : \tau, \Gamma) = x : \mathcal{T}(\tau), \mathbf{\Gamma}(\Gamma)$
$\mathcal{V}(x) = x$
$\mathcal{V}(v[\tau : \text{Type}]) = \mathcal{V}(v)[\mathcal{T}(\tau) : \text{Type}]$
$\mathcal{V}(v[\alpha : \text{Res}]) = \mathcal{V}(v)[\alpha : \text{Res}]$
$\mathcal{V}(v[U : \text{Cap}^1]) = \mathcal{V}(v)[\mathcal{U}(U) : \text{Cap}]$
$\mathcal{V}(v[A : \text{Cap}^+]) = \mathcal{V}(v)[\mathcal{A}(A) : \text{Cap}]$
$\mathcal{H}(\lambda\alpha{:}\kappa.h) = \lambda\alpha{:}\mathcal{K}(\kappa).\mathcal{H}(h)$
$\mathcal{H}(\lambda\alpha{:}\text{Cap}^+ \text{ where } U_1 \boxplus \alpha \leq A_1, \ldots, U_n \boxplus \alpha \leq A_n.h) =$
$\quad \lambda\alpha{:}\text{Cap}.[\alpha \leftarrow (\alpha\&(\mathcal{U}(U_1) \multimap \mathcal{A}(A_1) \otimes \text{true})\& \ldots \&(\mathcal{U}(U_n) \multimap \mathcal{A}(A_n) \otimes \text{true}))]\mathcal{H}(h)$
$\mathcal{H}(\lambda(C, x : \tau).e) = \lambda(\mathcal{C}(C), x : \mathcal{T}(\tau)).\mathcal{E}(e)$
$\mathcal{H}((v_1, v_2)) = (\mathcal{V}(v_1), \mathcal{V}(v_2))$
$\mathcal{D}(x = v) = (x = \mathcal{V}(v))$
$\mathcal{D}(x = h) = (x = \mathcal{H}(h))$
$\mathcal{D}(x = \#n\, v) = (x = \#n\, \mathcal{V}(v))$
$\mathcal{D}(\text{new}\, \alpha, x) = \text{new}\, \alpha, x$
$\mathcal{D}(\text{free}\, v) = \text{free}\, \mathcal{V}(v)$
$\mathcal{D}(\text{use}\, v) = \text{use}\, \mathcal{V}(v)$
$\mathcal{E}(\text{let}\, d \text{ in}\, e) = \text{let}\, \mathcal{D}(d) \text{ in}\, \mathcal{E}(e)$
$\mathcal{E}(v_1\ v_2) = \mathcal{V}(v_1)\ \mathcal{V}(v_2)$
$\mathcal{E}(\text{halt}) = \text{halt}$

C.3.1 Lemmas for CC2→LC

- If $\vdash \Delta$ and $\Delta \vdash U : \text{Cap}^1$ then $\mathbf{\Delta}(\Delta) \vdash [\![\Delta]\!]\mathcal{U}(U) : \text{Cap}$
- If $\vdash \Delta$ and $\Delta \vdash A : \text{Cap}^+$ then $\mathbf{\Delta}(\Delta) \vdash [\![\Delta]\!]\mathcal{A}(A) : \text{Cap}$
- If $\vdash \Delta$ and $\Delta \vdash \tau : \text{Type}$ then $\mathbf{\Delta}(\Delta) \vdash [\![\Delta]\!]\mathcal{T}(\tau) : \text{Type}$
- If $\vdash \Delta$ and $\Delta \vdash U_1 = U_2 : \text{Cap}^1$ then $\mathbf{\Delta}(\Delta) \vdash [\![\Delta]\!]\mathcal{U}(U_1) = [\![\Delta]\!]\mathcal{U}(U_2) : \text{Cap}$
- If $\vdash \Delta$ and $\Delta \vdash A_1 = A_2 : \text{Cap}^+$ then $\mathbf{\Delta}(\Delta) \vdash [\![\Delta]\!]\mathcal{A}(A_1) \otimes \text{true} = [\![\Delta]\!]\mathcal{A}(A_2) \otimes \text{true} : \text{Cap}$
- If $\vdash \Delta$ and $\Delta \vdash C \leq A$ then $[\![\Delta]\!]\mathcal{C}(C) \vdash [\![\Delta]\!]\mathcal{A}(A) \otimes \text{true}$
- If $\vdash \Delta$ and $\Delta \vdash \tau_1 = \tau_2 : \text{Type}$ then $\mathbf{\Delta}(\Delta) \vdash [\![\Delta]\!]\mathcal{T}(\tau_1) = [\![\Delta]\!]\mathcal{T}(\tau_2) : \text{Type}$
- If $\vdash \Delta$ and $\Delta \vdash \alpha' : \text{Cap}^+$ and $\Delta \vdash A : \text{Cap}^+$ then $\mathcal{A}([\alpha' \leftarrow A']A) = [\alpha' \leftarrow \mathcal{A}(A')]\mathcal{A}(A)$
- If $\vdash \Delta$ and $\Delta \vdash \alpha' : \text{Cap}^+$ and $\Delta \vdash U : \text{Cap}^1$ then $\mathcal{U}([\alpha' \leftarrow A']U) = [\alpha' \leftarrow A'']\mathcal{U}(U)$ for any A''
- If $\vdash \Delta$ and $\Delta \vdash \alpha' : \text{Cap}^1$ and $\Delta \vdash A : \text{Cap}^+$ then $\mathcal{A}([\alpha' \leftarrow U']A) = [\alpha' \leftarrow U'']\mathcal{A}(A)$ for any U''
- If $\vdash \Delta$ and $\Delta \vdash \alpha' : \text{Cap}^1$ and $\Delta \vdash U : \text{Cap}^1$ then $\mathcal{U}([\alpha' \leftarrow U']U) = [\alpha' \leftarrow \mathcal{U}(U')]\mathcal{U}(U)$
- If $\vdash \Delta$ and $\Delta \vdash \alpha' : \text{Cap}^+$ and $\Delta \vdash \tau : \text{Type}$ then $\mathcal{T}([\alpha' \leftarrow A']\tau) = [\alpha' \leftarrow \mathcal{A}(A')]\mathcal{T}(\tau)$

- If $\vdash \Delta$ and $\Delta \vdash \alpha' : \text{Cap}^1$ and $\Delta \vdash \tau : \text{Type}$ then $\mathcal{T}([\alpha' \leftarrow U']\tau) = [\alpha' \leftarrow \mathcal{U}(U')]\mathcal{T}(\tau)$
- If $\vdash \Delta$ and $\Delta \vdash \alpha' : \text{Type}$ and $\Delta \vdash \tau : \text{Type}$ then $\mathcal{T}([\alpha' \leftarrow \tau']\tau) = [\alpha' \leftarrow \mathcal{T}(\tau')]\mathcal{T}(\tau)$
- If $\vdash \Delta$ and $\Delta \vdash \Gamma$ then:
  - If $\Delta; \Gamma \vdash v : \tau$ then $\mathbf{\Delta}(\Delta); [\Delta]\mathbf{\Gamma}(\Gamma) \vdash [\Delta]\mathcal{V}(v) : [\Delta]\mathcal{T}(\tau)$
  - If $\Delta; \Gamma \vdash h : \tau$ then $\mathbf{\Delta}(\Delta); [\Delta]\mathbf{\Gamma}(\Gamma) \vdash [\Delta]\mathcal{H}(h) : [\Delta]\mathcal{T}(\tau)$
  - If $\Delta; \Gamma; C \vdash d \Longrightarrow \Delta'; \Gamma'; C'$ then $\mathbf{\Delta}(\Delta); [\Delta]\mathbf{\Gamma}(\Gamma); [\Delta]\mathcal{C}(C) \vdash [\Delta]\mathcal{D}(d) \Longrightarrow \mathbf{\Delta}(\Delta'); [\Delta]\mathbf{\Gamma}(\Gamma'); [\Delta]\mathcal{C}(C')$ and $[\Delta] = [\Delta']$
  - If $\Delta; \Gamma; C \vdash e$ then $\mathbf{\Delta}(\Delta); [\Delta]\mathbf{\Gamma}(\Gamma); [\Delta]\mathcal{C}(C) \vdash [\Delta]\mathcal{E}(e)$

C.3.2 Complete collection in extended LC

Extensions to LC for nonlinear capabilities:

$$
\begin{array}{rrcl}
\textit{types} & \tau & = & \ldots \mid\ !C \Rightarrow \tau \\
\textit{capabilities} & C & = & \ldots \mid C_1 \cup C_2 \mid\ !C \\
\textit{nonlinear cap ctxts} & [\Lambda] & = & [\,!C_1], \ldots, [\,!C_n] \\
\textit{cap ctxts} & \Lambda & = & C_1, \ldots, C_k, [\,!C_{k+1}], \ldots, [\,!C_n] \\
\\
\textit{word values} & v & = & \ldots \mid v[!] \\
\textit{heap values} & h & = & \ldots \mid \lambda\,!C.h
\end{array}
$$

Abbreviations:

$$
\begin{array}{rcl}
C_1 \times C_2 & = & !(C_1 \otimes C_2) \\
C_1 \Rightarrow C_2 & = & !(C_1 \multimap C_2) \\
C_1 \Leftrightarrow C_2 & = & (C_1 \Rightarrow C_2) \times (C_2 \Rightarrow C_1) \\
C_1 \vee C_2 & = & !(C_1 \cup C_2)
\end{array}
$$

New and modified rules:

$$
\frac{\Delta \vdash C_1 : \text{Cap} \qquad \Delta \vdash C_2 : \text{Cap}}{\Delta \vdash C_1 \cup C_2 : \text{Cap}} \qquad \frac{\Delta \vdash C : \text{Cap}}{\Delta \vdash !C : \text{Cap}} \qquad \frac{\Delta \vdash C : \text{Cap} \qquad \Delta \vdash \tau : \text{Type}}{\Delta \vdash !C \Rightarrow \tau}
$$

$$
[\Lambda], C \vdash C \qquad [\Lambda], [\,!C] \vdash C \qquad [\Lambda] \vdash \emptyset
$$

$$
\frac{\Lambda, [\,!C], [\,!C] \vdash C'}{\Lambda, [\,!C] \vdash C'} \qquad \frac{\Lambda \vdash C'}{\Lambda, [\,!C] \vdash C'}
$$

$$
\frac{[\Lambda] \vdash C}{[\Lambda] \vdash !C} \qquad \frac{\Lambda, [\,!C] \vdash C'}{\Lambda, \,!C \vdash C'}
$$

$$\frac{\Lambda \vdash C_k}{\Lambda \vdash C_1 \cup C_2}(k \in \{1,2\}) \qquad \frac{\Lambda, C_1 \vdash C_3 \quad \Lambda, C_2 \vdash C_3}{\Lambda, C_1 \cup C_2 \vdash C_3}$$

In $\Delta;\Gamma \vdash v : \tau$ and $\Delta;\Gamma \vdash h : \tau$ rules, replace $\Delta;\Gamma$ with $\Delta;\Gamma;[\Lambda]$.
In $\Delta;\Gamma;C \vdash d \Longrightarrow \Delta';\Gamma';C'$ rules, replace $\Delta;\Gamma;C$ with $\Delta;\Gamma;[\Lambda];C$.
In $\Delta;\Gamma;C \vdash e$ rules, replace $\Delta;\Gamma;C$ with $\Delta;\Gamma;[\Lambda];C$.

$$\frac{\Delta \vdash C_1 : \kappa \quad \Delta \vdash C_2 : \kappa \quad [\Lambda], C_1 \vdash C_2 \quad [\Lambda], C_2 \vdash C_1}{\Delta;[\Lambda] \vdash C_1 = C_2 : \kappa}$$

$$\frac{\Delta;\Gamma;[\Lambda] \vdash v : !C \Rightarrow \tau \quad [\Lambda] \vdash !C}{\Delta;\Gamma;[\Lambda] \vdash v[!] : \tau}$$

$$\frac{\Delta \vdash C : \mathrm{Cap} \quad \Delta;\Gamma;[\Lambda],[\,!C] \vdash h : \tau}{\Delta;\Gamma;[\Lambda] \vdash \lambda\,!C.h : !C \Rightarrow \tau}$$

$$\frac{[\Lambda], C \otimes \{\alpha\} \vdash C'}{\Delta;\Gamma;[\Lambda];C \vdash \mathrm{new}\,\alpha, x \Longrightarrow \Delta, \alpha : \mathrm{Res};\Gamma, x : \alpha\,\mathrm{handle};[\Lambda];C'}(\alpha \notin \mathit{domain}(\Delta)\;\mathit{and}\;x \notin \mathit{domain}(\Gamma))$$

$$\frac{\Delta;\Gamma;[\Lambda] \vdash v : \alpha\,\mathrm{handle} \quad [\Lambda], C \vdash C' \otimes \{\alpha\}}{\Delta;\Gamma;[\Lambda];C \vdash \mathrm{free}\,v \Longrightarrow \Delta;\Gamma;[\Lambda];C'}$$

$$\frac{\Delta;\Gamma;[\Lambda] \vdash v : \alpha\,\mathrm{handle} \quad [\Lambda], C \vdash \{\alpha\} \otimes \mathrm{true}}{\Delta;\Gamma;[\Lambda];C \vdash \mathrm{use}\,v \Longrightarrow \Delta;\Gamma;[\Lambda];C}$$

$$\frac{\Delta;\Gamma;[\Lambda] \vdash v_1 : (C', \tau) \to 0 \quad \Delta;\Gamma;[\Lambda] \vdash v_2 : \tau \quad [\Lambda], C \vdash C'}{\Delta;\Gamma;[\Lambda];C \vdash v_1\ v_2}$$

$$\frac{\Delta;[\Lambda] \vdash C = \emptyset : \mathrm{Cap}}{\Delta;\Gamma;[\Lambda];C \vdash \mathrm{halt}}$$

Modifications to CC2-to-LC translation:

$\mathcal{A}(\alpha) = \alpha$
$\mathcal{A}(\emptyset) = \emptyset$
$\mathcal{A}(\{\rho^\varphi\}) = \{\rho\}$
$\mathcal{A}(A_1 \oplus A_2) = (\mathcal{A}(A_1) \otimes \mathcal{Z}(A_1 \oplus A_2))\&(\mathcal{A}(A_2) \otimes \mathcal{Z}(A_1 \oplus A_2))$
$\mathcal{Z}(\alpha) = \alpha_Z$
$\mathcal{Z}(\emptyset) = \emptyset$
$\mathcal{Z}(\{\rho^\varphi\}) = \mathrm{true}$
$\mathcal{Z}(A_1 \oplus A_2) = \mathcal{Z}(A_1) \otimes \mathcal{Z}(A_2)$
$\mathcal{C}(U \boxplus A) = \mathcal{U}(U) \otimes (\mathcal{A}(A) \otimes \mathcal{Z}(A))$
$\mathcal{S}(U_1 \boxplus A_1 \leq A_2) = (\mathcal{C}(U_1 \boxplus A_1) \Rightarrow \mathcal{A}(A_2) \otimes \mathcal{Z}(A_2)) \times (\mathcal{U}(U_1) \cup \mathcal{Z}(A_1) \Rightarrow \mathcal{Z}(A_2))$
$\mathcal{T}(\forall\alpha{:}\mathrm{Cap}^+.\tau) = \forall\alpha{:}\mathrm{Cap}.\forall\alpha_Z{:}\mathrm{Cap}.(((\alpha \Leftrightarrow \emptyset) \times (\alpha_Z \Leftrightarrow \emptyset)) \vee (\alpha_Z \Leftrightarrow \mathrm{true})) \Rightarrow \mathcal{T}(\tau)$
$\mathcal{T}(\forall\alpha{:}\mathrm{Cap}^+\ \mathrm{where}\ C_1 \leq A_1, \ldots, C_n \leq A_n.\tau) =$
$\quad\forall\alpha{:}\mathrm{Cap}.\forall\alpha_Z{:}\mathrm{Cap}.((((\alpha \Leftrightarrow \emptyset) \times (\alpha_Z \Leftrightarrow \emptyset)) \vee (\alpha_Z \Leftrightarrow \mathrm{true}))\times$
$\quad\mathcal{S}(C_1 \leq A_1) \times \ldots \times \mathcal{S}(C_n \leq A_n)) \Rightarrow \mathcal{T}(\tau)$
$\mathbf{\Delta}(\alpha : \mathrm{Cap}^+, \Delta) = \alpha : \mathrm{Cap}, \alpha_Z : \mathrm{Cap}, \mathbf{\Delta}(\Delta)$
$\mathbf{\Delta}(\alpha{:}\mathrm{Cap}^+\ \mathrm{where}\ U_1 \boxplus \alpha \leq A_1, \ldots, U_n \boxplus \alpha \leq A_n, \Delta) = \alpha : \mathrm{Cap}, \alpha_Z : \mathrm{Cap}, \mathbf{\Delta}(\Delta)$
$[\Delta] = []$

$\mathbf{\Lambda}(\cdot) = \cdot$
$\mathbf{\Lambda}(\alpha : \kappa, \Delta) = \mathbf{\Lambda}(\Delta)\, \mathit{where}\, \kappa \neq \mathrm{Cap}^+$
$\mathbf{\Lambda}(\alpha : \mathrm{Cap}^+, \Delta) = [((\alpha \Leftrightarrow \emptyset) \times (\alpha_Z \Leftrightarrow \emptyset)) \vee (\alpha_Z \Leftrightarrow \mathrm{true})], \mathbf{\Lambda}(\Delta)$
$\mathbf{\Lambda}(\alpha{:}\mathrm{Cap}^+ \,\mathrm{where}\, C_1 \leq A_1, \ldots, C_n \leq A_n, \Delta) =$
$\quad [(((\alpha \Leftrightarrow \emptyset) \times (\alpha_Z \Leftrightarrow \emptyset)) \vee (\alpha_Z \Leftrightarrow \mathrm{true}))\times$
$\quad \mathcal{S}(C_1 \leq A_1) \times \ldots \times \mathcal{S}(C_n \leq A_n)], \mathbf{\Lambda}(\Delta)$
$\mathcal{V}(v[A : \mathrm{Cap}^+]) = \mathcal{V}(v)[\mathcal{A}(A) : \mathrm{Cap}][\mathcal{Z}(A) : \mathrm{Cap}][!]$
$\mathcal{H}(\lambda\alpha{:}\mathrm{Cap}^+.h) = \lambda\alpha{:}\mathrm{Cap}.\lambda\alpha_Z{:}\mathrm{Cap}.$
$\quad \lambda(((\alpha \Leftrightarrow \emptyset) \times (\alpha_Z \Leftrightarrow \emptyset)) \vee (\alpha_Z \Leftrightarrow \mathrm{true})).\mathcal{H}(h)$
$\mathcal{H}(\lambda\alpha{:}\mathrm{Cap}^+ \,\mathrm{where}\, C_1 \leq A_1, \ldots, C_n \leq A_n.h) = \lambda\alpha{:}\mathrm{Cap}.\lambda\alpha_Z{:}\mathrm{Cap}.$
$\quad \lambda((((\alpha \Leftrightarrow \emptyset) \times (\alpha_Z \Leftrightarrow \emptyset)) \vee (\alpha_Z \Leftrightarrow \mathrm{true}))\times$
$\quad \mathcal{S}(C_1 \leq A_1) \times \ldots \times \mathcal{S}(C_n \leq A_n)).\mathcal{H}(h)$

- If $\vdash \Delta$ and $\Delta \vdash U : \mathrm{Cap}^1$ then $\mathbf{\Delta}(\Delta) \vdash \mathcal{U}(U) : \mathrm{Cap}$
- If $\vdash \Delta$ and $\Delta \vdash A : \mathrm{Cap}^+$ then $\mathbf{\Delta}(\Delta) \vdash \mathcal{A}(A) : \mathrm{Cap}$ and $\mathbf{\Delta}(\Delta) \vdash \mathcal{Z}(A) : \mathrm{Cap}$
- If $\vdash \Delta$ and $\Delta \vdash A : \mathrm{Cap}^+$ then $\mathbf{\Lambda}(\Delta) \vdash ((\mathcal{A}(A) \Leftrightarrow \emptyset) \times (\mathcal{Z}(A) \Leftrightarrow \emptyset)) \vee (\mathcal{Z}(A) \Leftrightarrow \mathrm{true})$
- If $\vdash \Delta$ and $\Delta \vdash \tau : \mathrm{Type}$ then $\mathbf{\Delta}(\Delta) \vdash \mathcal{T}(\tau) : \mathrm{Type}$
- If $\vdash \Delta$ and $\Delta \vdash U_1 = U_2 : \mathrm{Cap}^1$ then $\mathbf{\Delta}(\Delta); \mathbf{\Lambda}(\Delta) \vdash \mathcal{U}(U_1) = \mathcal{U}(U_2) : \mathrm{Cap}$
- If $\vdash \Delta$ and $\Delta \vdash A_1 = A_2 : \mathrm{Cap}^+$ then $\mathbf{\Delta}(\Delta); \mathbf{\Lambda}(\Delta) \vdash \mathcal{A}(A_1) \otimes \mathcal{Z}(A_1) = \mathcal{A}(A_2) \otimes \mathcal{Z}(A_2) : \mathrm{Cap}$
- If $\vdash \Delta$ and $\Delta \vdash U \boxplus A \leq A'$ then $\mathbf{\Lambda}(\Delta), \mathcal{U}(U) \cup \mathcal{Z}(A) \vdash \mathcal{Z}(A')$
- If $\vdash \Delta$ and $\Delta \vdash C \leq A$ then $\mathbf{\Lambda}(\Delta), \mathcal{C}(C) \vdash \mathcal{A}(A) \otimes \mathcal{Z}(A)$
- If $\vdash \Delta$ and $\Delta \vdash \tau_1 = \tau_2 : \mathrm{Type}$ then $\mathbf{\Delta}(\Delta); \mathbf{\Lambda}(\Delta) \vdash \mathcal{T}(\tau_1) = \mathcal{T}(\tau_2) : \mathrm{Type}$
- If $\vdash \Delta$ and $\Delta \vdash \alpha' : \mathrm{Cap}^+$ and $\Delta \vdash A : \mathrm{Cap}^+$ then $\mathcal{A}([\alpha' \leftarrow A']A) = [\alpha' \leftarrow \mathcal{A}(A'), \alpha'_Z \leftarrow \mathcal{Z}(A')]\mathcal{A}(A)$
- If $\vdash \Delta$ and $\Delta \vdash \alpha' : \mathrm{Cap}^+$ and $\Delta \vdash A : \mathrm{Cap}^+$ then $\mathcal{Z}([\alpha' \leftarrow A']A) = [\alpha' \leftarrow \mathcal{A}(A'), \alpha'_Z \leftarrow \mathcal{Z}(A')]\mathcal{Z}(A)$
- If $\vdash \Delta$ and $\Delta \vdash \alpha' : \mathrm{Cap}^+$ and $\Delta \vdash U : \mathrm{Cap}^1$ then $\mathcal{U}([\alpha' \leftarrow A']U) = [\alpha' \leftarrow A'', \alpha'_Z \leftarrow A''_Z]\mathcal{U}(U)$ for any A'', A''_Z
- If $\vdash \Delta$ and $\Delta \vdash \alpha' : \mathrm{Cap}^1$ and $\Delta \vdash A : \mathrm{Cap}^+$ then $\mathcal{A}([\alpha' \leftarrow U']A) = [\alpha' \leftarrow U'']\mathcal{A}(A)$ for any U''
- If $\vdash \Delta$ and $\Delta \vdash \alpha' : \mathrm{Cap}^1$ and $\Delta \vdash A : \mathrm{Cap}^+$ then $\mathcal{Z}([\alpha' \leftarrow U']A) = [\alpha' \leftarrow U'']\mathcal{Z}(A)$ for any U''
- If $\vdash \Delta$ and $\Delta \vdash \alpha' : \mathrm{Cap}^1$ and $\Delta \vdash U : \mathrm{Cap}^1$ then $\mathcal{U}([\alpha' \leftarrow U']U) = [\alpha' \leftarrow \mathcal{U}(U')]\mathcal{U}(U)$
- If $\vdash \Delta$ and $\Delta \vdash \alpha' : \mathrm{Cap}^+$ and $\Delta \vdash \tau : \mathrm{Type}$ then $\mathcal{T}([\alpha' \leftarrow A']\tau) = [\alpha' \leftarrow \mathcal{A}(A'), \alpha'_Z \leftarrow \mathcal{Z}(A')]\mathcal{T}(\tau)$
- If $\vdash \Delta$ and $\Delta \vdash \alpha' : \mathrm{Cap}^1$ and $\Delta \vdash \tau : \mathrm{Type}$ then $\mathcal{T}([\alpha' \leftarrow U']\tau) = [\alpha' \leftarrow \mathcal{U}(U')]\mathcal{T}(\tau)$
- If $\vdash \Delta$ and $\Delta \vdash \alpha' : \mathrm{Type}$ and $\Delta \vdash \tau : \mathrm{Type}$ then $\mathcal{T}([\alpha' \leftarrow \tau']\tau) = [\alpha' \leftarrow \mathcal{T}(\tau')]\mathcal{T}(\tau)$
- If $\vdash \Delta$ and $\Delta \vdash \Gamma$ then:
  - If $\Delta; \Gamma \vdash v : \tau$ then $\mathbf{\Delta}(\Delta); \mathbf{\Gamma}(\Gamma); \mathbf{\Lambda}(\Delta) \vdash \mathcal{V}(v) : \mathcal{T}(\tau)$
  - If $\Delta; \Gamma \vdash h : \tau$ then $\mathbf{\Delta}(\Delta); \mathbf{\Gamma}(\Gamma); \mathbf{\Lambda}(\Delta) \vdash \mathcal{H}(h) : \mathcal{T}(\tau)$
  - If $\Delta; \Gamma; C \vdash d \Longrightarrow \Delta'; \Gamma'; C'$ then $\mathbf{\Delta}(\Delta); \mathbf{\Gamma}(\Gamma); \mathbf{\Lambda}(\Delta); \mathcal{C}(C) \vdash \mathcal{D}(d) \Longrightarrow \mathbf{\Delta}(\Delta'); \mathbf{\Gamma}(\Gamma'); \mathbf{\Lambda}(\Delta'); \mathcal{C}(C')$
  - If $\Delta; \Gamma; C \vdash e$ then $\mathbf{\Delta}(\Delta); \mathbf{\Gamma}(\Gamma); \mathbf{\Lambda}(\Delta); \mathcal{C}(C) \vdash \mathcal{E}(e)$

C.4 Translation: CC2→linear $F\omega$

$\mathcal{K}(\mathrm{Cap}^1) = \mathrm{Type}$
$\mathcal{K}(\mathrm{Cap}^+) = \mathrm{Type} \to \mathrm{Type}$
$\mathcal{K}(\mathrm{Type}) = \mathrm{Type}$
$\mathcal{K}(\mathrm{Res}) = \mathrm{Type}$
$\mathcal{U}(\alpha) = \alpha$
$\mathcal{U}(\emptyset) = ()$
$\mathcal{U}(\{\alpha^\varphi\}) = \chi\ \alpha$
$\mathcal{U}(U_1 \oplus U_2) = \mathcal{U}(U_1) \otimes \mathcal{U}(U_2)$
$\mathcal{A}(A) = \lambda\gamma{:}\mathrm{Type}.\exists\delta{:}\mathrm{Type}.(\gamma \rightrightarrows \delta)\times !(\mathcal{A}[A]\ \delta)$
$\mathcal{A}[\alpha] = \alpha$
$\mathcal{A}[\emptyset] = \lambda\gamma{:}\mathrm{Type}.(\circ)$
$\mathcal{A}[\{\alpha^\varphi\}] = \lambda\gamma{:}\mathrm{Type}.\gamma \rightrightarrows\!\!\circ\ \chi\ \alpha$
$\mathcal{A}[A_1 \oplus A_2] = \lambda\gamma{:}\mathrm{Type}.\,!(\mathcal{A}(A_1)\ \gamma)\otimes !(\mathcal{A}(A_2)\ \gamma)$
$\mathcal{C}(U \boxplus A) = \lambda\gamma{:}\mathrm{Type}.\mathcal{U}(U)\otimes !(\mathcal{A}(A)\ \gamma)$
$\mathcal{S}(U \boxplus A \leq A') = \forall\delta{:}\mathrm{Type}.\,!(\mathcal{A}(A)\ \delta) \to !(\mathcal{A}(A')\ (\delta \otimes \mathcal{U}(U)))$
$\mathcal{T}(\alpha) = \alpha$
$\mathcal{T}(\tau_1 \times \tau_2) = !\mathcal{T}(\tau_1)\otimes !\mathcal{T}(\tau_2)$
$\mathcal{T}((U \boxplus A, \tau) \to 0) = \forall\gamma{:}\mathrm{Type}.\gamma \to \mathcal{U}(U) \multimap !(\mathcal{A}(A)\ \gamma) \multimap !\mathcal{T}(\tau) \multimap \mathrm{true}$
$\mathcal{T}(\alpha\,\mathrm{handle}) = (\circ)$
$\mathcal{T}(\forall\alpha{:}\kappa.\tau) = \forall\alpha{:}\mathcal{K}(\kappa).\,!\mathcal{T}(\tau)$
$\mathcal{T}(\forall\alpha{:}\mathrm{Cap}^+\,\mathrm{where}\,C_1 \leq A_1, \ldots, C_n \leq A_n.\tau) =$
$\quad\forall\alpha{:}\mathrm{Type} \to \mathrm{Type}.\,!\mathcal{S}(C_1 \leq A_1) \to \ldots \to !\mathcal{S}(C_n \leq A_n) \to !\mathcal{T}(\tau)$
$\mathbf{\Delta}(\cdot) = \cdot$
$\mathbf{\Delta}(\alpha : \kappa, \Delta) = \alpha : \mathcal{K}(\kappa), \mathbf{\Delta}(\Delta)$
$\mathbf{\Delta}(\alpha{:}\mathrm{Cap}^+\,\mathrm{where}\,U_1 \boxplus \alpha \leq A_1, \ldots, U_n \boxplus \alpha \leq A_n, \Delta) = \alpha : \mathrm{Type} \to \mathrm{Type}, \mathbf{\Delta}(\Delta)$
$\mathbf{\Gamma}(\cdot;\cdot) = \cdot$
$\mathbf{\Gamma}(\cdot; x : \tau, \Gamma) = !(x : \mathcal{T}(\tau)), \mathbf{\Gamma}(\cdot;\Gamma)$
$\mathbf{\Gamma}(\alpha : \kappa, \Delta; \Gamma) = \mathbf{\Gamma}(\Delta;\Gamma)$
$\mathbf{\Gamma}(\alpha{:}\mathrm{Cap}^+\,\mathrm{where}\,(C_1 \leq A_1, \ldots, C_n \leq A_n), \Delta; \Gamma) =$
$\quad!(x_{A1} : \mathcal{S}(C_1 \leq A_1)), \ldots, !(x_{An} : \mathcal{S}(C_n \leq A_n)), \mathbf{\Gamma}(\Delta;\Gamma)$

The translations of v, h, and e are directed by typing judgments:

- $\Delta;\Gamma \vdash v : \tau \rightsquigarrow \mathcal{V}(v)$
- $\Delta;\Gamma \vdash h : \tau \rightsquigarrow \mathcal{H}(h)$
- $\Delta;\Gamma;C \vdash e \rightsquigarrow \mathcal{E}(e)$

For conciseness, though, some of the definitions below suppress the typing judgment when it is not immediately relevant.

$\mathcal{V}(x) = x$
$\mathcal{V}(v[\tau : \mathrm{Type}]) = \mathcal{V}(v)\ \mathcal{T}(\tau)$
$\mathcal{V}(v[\alpha : \mathrm{Res}]) = \mathcal{V}(v)\ \alpha$
$\mathcal{V}(v[U : \mathrm{Cap}^1]) = \mathcal{V}(v)\ \mathcal{U}(U)$
$\mathcal{V}(v[A : \mathrm{Cap}^+]) = \mathcal{V}(v)\ \mathcal{A}(A)$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash v : \forall\alpha{:}\mathrm{Cap}^+\,\mathrm{where}\,U_1 \boxplus \alpha \leq A_1, \ldots, U_n \boxplus \alpha \leq A_n.\tau \rightsquigarrow e \\ \Delta \vdash U_1 \boxplus \alpha \leq A_1 \rightsquigarrow e_1 \qquad \ldots \qquad \Delta \vdash U_n \boxplus \alpha \leq A_n \rightsquigarrow e_n \\ \Delta;\Gamma \vdash A : \mathrm{Cap}^+\end{array}}{\Delta;\Gamma \vdash v[A : \mathrm{Cap}^+] : [\alpha \leftarrow A]\tau \rightsquigarrow e\ \mathcal{A}[A]\ e_1\ \ldots e_n}$$

$$\frac{\Delta;\Gamma \vdash v : \tau' \rightsquigarrow e \qquad \Delta \vdash \tau' = \tau : \mathrm{Type} \rightsquigarrow e'}{\Delta;\Gamma \vdash v : \tau \rightsquigarrow (\#1\,e')\ e}$$

$\mathcal{H}(\lambda\alpha{:}\kappa.h) = !\lambda\alpha{:}\mathcal{K}(\kappa).\mathcal{H}(h)$
$\mathcal{H}(\lambda\alpha{:}\mathrm{Cap}^{+}\ \mathrm{where}\ C_1 \leq A_1, \ldots, C_n \leq A_n.h) =$
$\quad !\lambda\alpha{:}\mathrm{Type} \rightarrow \mathrm{Type}.\, !\lambda\, !x_{A1}{:}!\mathcal{S}(C_1 \leq A_1). \ldots !\lambda\, !x_{An}{:}!\mathcal{S}(C_n \leq A_n).\mathcal{H}(h)$
$\mathcal{H}(\lambda(U \boxplus A, x : \tau).e) = !\lambda\gamma{:}\mathrm{Type}.\, !\lambda z_P{:}\gamma.\lambda z_U{:}\mathcal{U}(U).\lambda\, !z_A{:}!(\mathcal{A}(A)\ \gamma).\lambda\, !x{:}!\mathcal{T}(\tau).\mathcal{E}(e)$
$\mathcal{H}((v_1, v_2)) = (\mathcal{V}(v_1), \mathcal{V}(v_1))$

$$\frac{\Delta; \Gamma \vdash h : \tau' \rightsquigarrow e \qquad \Delta \vdash \tau' = \tau : \mathrm{Type} \rightsquigarrow e'}{\Delta; \Gamma \vdash h : \tau \rightsquigarrow (\#1\, e')\ e}$$

$\mathcal{E}(\mathrm{let}\, x = v\ \mathrm{in}\, e) = \mathrm{let}\ !x = \mathcal{V}(v)\ \mathrm{in}\, \mathcal{E}(e)$
$\mathcal{E}(\mathrm{let}\, x = h\ \mathrm{in}\, e) = \mathrm{let}\ !x = \mathcal{H}(h)\ \mathrm{in}\, \mathcal{E}(e)$
$\mathcal{E}(\mathrm{let}\, x = \#n\, v\ \mathrm{in}\, e) = \mathrm{let}\ !x = \#n\, \mathcal{V}(v)\ \mathrm{in}\, \mathcal{E}(e)$
$\mathcal{E}(\mathrm{let\, new}\, \alpha, x\ \mathrm{in}\, e) =$
$\quad \mathrm{let}\, [\alpha, x_A] = \mathit{new}\ ()\ \mathrm{in}$
$\quad \mathrm{let}\, z_U = \langle z_U, x_A \rangle\ \mathrm{in}$
$\quad \mathrm{let}\ !x = ()\ \mathrm{in}$
$\quad \mathcal{E}(e)$

$$\frac{\Delta; \Gamma \vdash v : \alpha\, \mathrm{handle} \qquad \Delta \vdash U = U' \oplus \{\alpha^1\} : \mathrm{Cap}^1 \rightsquigarrow e_1 \qquad \Delta; \Gamma; U' \boxplus A \vdash e \rightsquigarrow e'}{\Delta; \Gamma; U \boxplus A \vdash \mathrm{let\, free}\, v\ \mathrm{in}\, e \rightsquigarrow \mathit{expfree}}$$

$\mathit{expfree} =$
$\quad \mathrm{let}\, \langle z_U, x_A \rangle = (\#1\, e_1)\ z_U\ \mathrm{in}$
$\quad \mathrm{let}\ _ = \mathit{free}\ \alpha\ x_A\ \mathrm{in}$
$\quad e'$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash v : \alpha\, \mathrm{handle} \qquad \Delta \vdash U = U_B \oplus U' : \mathrm{Cap}^1 \rightsquigarrow e_1 \\ \Delta \vdash U_B \boxplus A \leq A' \oplus \{\alpha^+\} \rightsquigarrow e_2 \qquad \Delta; \Gamma; U \boxplus A \vdash e \rightsquigarrow e'\end{array}}{\Delta; \Gamma; U \boxplus A \vdash \mathrm{let\, use}\, v\ \mathrm{in}\, e \rightsquigarrow \mathit{expuse}}$$

$\mathit{expuse} =$
$\quad \mathrm{let}\, \langle z_{U1}, z_{U2} \rangle = (\#1\, e_1)\ z_U\ \mathrm{in}$
$\quad \mathrm{let}\ !z'_A = e_2\ \gamma\ z_A\ \mathrm{in}$
$\quad \mathrm{let}\, [\delta, \langle\, !x_{PQ},\, !z''_A \rangle] = z'_A\ \mathrm{in}$
$\quad \mathrm{let}\, \langle _,\, !z'''_A \rangle = z''_A\ \mathrm{in}$
$\quad \mathrm{let}\, [\epsilon, \langle\, !x_{QR},\, !x_{RA} \rangle] = z'''_A\ \mathrm{in}$
$\quad \mathrm{let}\ !x_{PR} = \mathit{etrans}\ x_{PQ}\ x_{QR}\ \mathrm{in}$
$\quad \mathrm{let}\ !x_{PA} = \mathit{etrans}\ x_{PR}\ x_{RA}\ \mathrm{in}$
$\quad \mathrm{let}\, \langle x_A, x_{AP} \rangle = x_{PA}\ \langle z_P, z_{U1} \rangle\ \mathrm{in}$
$\quad \mathrm{let}\, x_A = \mathit{use}\ \alpha\ x_A\ \mathrm{in}$
$\quad \mathrm{let}\, \langle z_P, z_{U1} \rangle = x_{AP}\ x_A\ \mathrm{in}$
$\quad \mathrm{let}\, z_U = (\#2\, e_1)\ \langle z_{U1}, z_{U2} \rangle\ \mathrm{in}$
$\quad e'$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash v_1 : (U' \boxplus A', \tau) \rightarrow 0 \rightsquigarrow e_1 \qquad \Delta; \Gamma \vdash v_2 : \tau \rightsquigarrow e_2 \\ \Delta \vdash U = U_B \oplus U' : \mathrm{Cap}^1 \rightsquigarrow e'_1 \qquad \Delta \vdash U_B \boxplus A \leq A' \rightsquigarrow e'_2\end{array}}{\Delta; \Gamma; U \boxplus A \vdash v_1\ v_2 \rightsquigarrow \mathit{expcall}}$$

$\mathit{expcall} =$
$\quad \mathrm{let}\, \langle z_{U2}, z'_U \rangle = (\#1\, e'_1)\ z_U\ \mathrm{in}$
$\quad \mathrm{let}\ !z'_A = e'_2\ \gamma\ z_A\ \mathrm{in}$
$\quad e_1\ \gamma \otimes u_2\ \langle z_P, z_{U2} \rangle\ z'_U\ z'_A\ e_2$

$$\frac{\Delta \vdash U = \emptyset : \mathrm{Cap}^1 \rightsquigarrow e_1 \qquad \Delta \vdash A = \emptyset : \mathrm{Cap}^+ \rightsquigarrow e_2}{\Delta; \Gamma; U \boxplus A \vdash \mathrm{halt} \rightsquigarrow \mathrm{let}\ _ = (\#1\, e_1)\ z_U\ \mathrm{in\, pack}[\gamma, z_P]\ \mathrm{as\, true}}$$

$$\frac{\Delta \vdash Q_1 = Q_1' : \kappa \rightsquigarrow e_1 \qquad \Delta \vdash Q_2 = Q_2' : \kappa \rightsquigarrow e_2}{\Delta \vdash Q_1 \oplus Q_2 = Q_1' \oplus Q_2' : \kappa \rightsquigarrow \mathit{ilprod}\ e_1\ e_2}(\kappa = \mathrm{Cap}^1)$$

$$\frac{\Delta \vdash Q : \kappa}{\Delta \vdash \emptyset \oplus Q = Q : \kappa \rightsquigarrow \mathit{ilprodunitleftleft}}(\kappa = \mathrm{Cap}^1)$$

$$\frac{\Delta \vdash Q_1 : \kappa \qquad \Delta \vdash Q_2 : \kappa}{\Delta \vdash Q_1 \oplus Q_2 = Q_2 \oplus Q_1 : \kappa \rightsquigarrow \mathit{ilprodcomm}}(\kappa = \mathrm{Cap}^1)$$

$$\frac{\Delta \vdash Q_1 : \kappa \qquad \Delta \vdash Q_2 : \kappa \qquad \Delta \vdash Q_3 : \kappa}{\Delta \vdash (Q_1 \oplus Q_2) \oplus Q_3 = Q_1 \oplus (Q_2 \oplus Q_3) : \kappa \rightsquigarrow \mathit{isymm}\ \mathit{ilprodassoc}}(\kappa = \mathrm{Cap}^1)$$

$$\frac{\Delta \vdash Q_1 = Q_1' : \kappa \rightsquigarrow e_1 \qquad \Delta \vdash Q_2 = Q_2' : \kappa \rightsquigarrow e_2}{\Delta \vdash Q_1 \oplus Q_2 = Q_1' \oplus Q_2' : \kappa \rightsquigarrow \mathit{aeqcong}}(\kappa = \mathrm{Cap}^+)$$

$\mathit{aeqcong} = !\lambda\gamma{:}\mathrm{Type}.\mathit{iexist}\ (\,!(\lambda\delta{:}\mathrm{Type}.\mathit{iprod}\ \mathit{irefl}\ (\mathit{iprod}\ (e_1\ \delta)\ (e_2\ \delta))))$

$$\frac{\Delta \vdash Q : \kappa}{\Delta \vdash \emptyset \oplus Q = Q : \kappa \rightsquigarrow \mathit{aequnit}}(\kappa = \mathrm{Cap}^+)$$

$\mathit{aequnit} = !\lambda\gamma{:}\mathrm{Type}.\,!\langle$
$\quad !\lambda[\delta, \langle\, !x_{PQ}, \langle _, [\epsilon, \langle\, !x_{QR},\, !x\rangle]\rangle\rangle] {:}!\mathcal{A}(\emptyset \oplus A)\ \gamma.$
$\qquad !\mathrm{pack}[\epsilon,\ !\langle \mathit{etrans}\ x_{PQ}\ x_{QR}, x\rangle]\ \mathrm{as}\ \mathcal{A}(A)\ \gamma,$
$\quad !\lambda\, !x {:}!\mathcal{A}(A)\ \gamma.$
$\qquad !\mathrm{pack}[\gamma,\ !\langle \mathit{erefl},\ !\langle\, !\mathrm{pack}[\gamma,\ !\langle \mathit{erefl}, ()\rangle]\ \mathrm{as}\ \mathcal{A}(\emptyset)\ \gamma, x\rangle\rangle]\ \mathrm{as}\ \mathcal{A}(\emptyset \oplus A)\ \gamma\rangle$

$$\frac{\Delta \vdash Q_1 : \kappa \qquad \Delta \vdash Q_2 : \kappa}{\Delta \vdash Q_1 \oplus Q_2 = Q_2 \oplus Q_1 : \kappa \rightsquigarrow \mathit{aeqcomm}}(\kappa = \mathrm{Cap}^+)$$

$\mathit{aeqcomm} = !\lambda\gamma{:}\mathrm{Type}.\mathit{iexist}\ (\,!(\lambda\delta{:}\mathrm{Type}.\mathit{iprod}\ \mathit{irefl}\ \mathit{iprodcomm}))$

$$\frac{\Delta \vdash Q_1 : \kappa \qquad \Delta \vdash Q_2 : \kappa \qquad \Delta \vdash Q_3 : \kappa}{\Delta \vdash (Q_1 \oplus Q_2) \oplus Q_3 = Q_1 \oplus (Q_2 \oplus Q_3) : \kappa \rightsquigarrow \mathit{aeqassoc}}(\kappa = \mathrm{Cap}^+)$$

$\mathit{aeqassoc} = !\lambda\gamma{:}\mathrm{Type}.\,!\langle$
$\ !\lambda[\delta, \langle\, !x_{PQ}, \langle\, !x_{12},\, !x_3\rangle\rangle] {:}!\mathcal{A}(A_1 \oplus A_2 \oplus A_3)\ \gamma.$
$\quad \mathrm{let}\ [\epsilon, \langle\, !x_{QR}, \langle\, !x_1,\, !x_2\rangle\rangle] = x_{12}\ \mathrm{in}$
$\quad \mathrm{let}\ [\eta, \langle\, !x_{RS},\, !x_{A2}\rangle] = x_2\ \mathrm{in}$
$\quad \mathrm{let}\ [\eta', \langle\, !x'_{RS},\, !x_{A1}\rangle] = x_1\ \mathrm{in}$
$\quad \mathrm{let}\ !y_1 = !\mathrm{pack}[\eta',\ !\langle \mathit{etrans}\ x_{QR}\ x'_{RS}, x_{A1}\rangle]\ \mathrm{as}\ \mathcal{A}(A_1)\ \delta\ \mathrm{in}$
$\quad \mathrm{let}\ !y_2 = !\mathrm{pack}[\eta,\ !\langle \mathit{etrans}\ x_{QR}\ x_{RS}, x_{A2}\rangle]\ \mathrm{as}\ \mathcal{A}(A_2)\ \delta\ \mathrm{in}$
$\quad \mathrm{let}\ !y_{23} = !\mathrm{pack}[\delta,\ !\langle \mathit{erefl},\ !\langle y_2, x_3\rangle\rangle]\ \mathrm{as}\ \mathcal{A}(A_2 \oplus A_3)\ \delta\ \mathrm{in}$
$\quad !\mathrm{pack}[\delta,\ !\langle x_{PQ},\ !\langle y_1, y_{23}\rangle\rangle]\ \mathrm{as}\ \mathcal{A}(A_1 \oplus (A_2 \oplus A_3))\ \gamma,$
$\ !\lambda[\delta, \langle\, !x_{PQ}, \langle\, !x_1,\, !x_{23}\rangle\rangle] {:}!\mathcal{A}(A_1 \oplus (A_2 \oplus A_3))\ \gamma.$
$\quad \mathrm{let}\ [\epsilon, \langle\, !x_{QR}, \langle\, !x_2,\, !x_3\rangle\rangle] = x_{23}\ \mathrm{in}$
$\quad \mathrm{let}\ [\eta, \langle\, !x_{RS},\, !x_{A2}\rangle] = x_2\ \mathrm{in}$
$\quad \mathrm{let}\ [\eta', \langle\, !x'_{RS},\, !x_{A3}\rangle] = x_3\ \mathrm{in}$
$\quad \mathrm{let}\ !y_3 = !\mathrm{pack}[\eta',\ !\langle \mathit{etrans}\ x_{QR}\ x'_{RS}, x_{A3}\rangle]\ \mathrm{as}\ \mathcal{A}(A_3)\ \delta\ \mathrm{in}$
$\quad \mathrm{let}\ !y_2 = !\mathrm{pack}[\eta,\ !\langle \mathit{etrans}\ x_{QR}\ x_{RS}, x_{A2}\rangle]\ \mathrm{as}\ \mathcal{A}(A_2)\ \delta\ \mathrm{in}$
$\quad \mathrm{let}\ !y_{12} = !\mathrm{pack}[\delta,\ !\langle \mathit{erefl},\ !\langle x_1, y_2\rangle\rangle]\ \mathrm{as}\ \mathcal{A}(A_1 \oplus A_2)\ \delta\ \mathrm{in}$
$\quad !\mathrm{pack}[\delta,\ !\langle x_{PQ},\ !\langle y_{12}, y_3\rangle\rangle]\ \mathrm{as}\ \mathcal{A}(A_1 \oplus A_2 \oplus A_3)\ \gamma\rangle$

$$\frac{\Delta \vdash A : \mathrm{Cap}^+}{\Delta \vdash A = A \oplus A : \mathrm{Cap}^+ \rightsquigarrow \mathit{aeqidem}}$$

$\mathit{aeqidem} = !\lambda\gamma{:}\mathrm{Type}.\,!\langle$
$\quad !\lambda\,!x{:}!\mathcal{A}(A)\ \gamma.$
$\qquad !\mathrm{pack}[\gamma,\ !\langle \mathit{erefl},\ !\langle x, x\rangle\rangle]\,\mathrm{as}\,\mathcal{A}(A \oplus A)\ \gamma,$
$\quad !\lambda[\delta, \langle\, !x_{PQ}, \langle _, [\epsilon, \langle\, !x_{QR},\ !x\rangle]\rangle\rangle]\,{:}!\mathcal{A}(A \oplus A)\ \gamma.$
$\qquad !\mathrm{pack}[\epsilon,\ !\langle \mathit{etrans}\ x_{PQ}\ x_{QR}, x\rangle]\,\mathrm{as}\,\mathcal{A}(A)\ \gamma\rangle$

$$\ldots, (\alpha{:}\mathrm{Cap}^+\ \text{where}\ \ldots, U_k \boxplus \alpha \leq A_k, \ldots), \ldots \vdash U_k \boxplus \alpha \leq A_k \rightsquigarrow x_{Ak}$$

$$\frac{\Delta \vdash \alpha : \mathrm{Res}}{\Delta \vdash \{\alpha^1\} \boxplus \emptyset \leq \{\alpha^+\} \rightsquigarrow \mathit{subres}}$$

$\mathit{subres} =$
$\qquad !\lambda\gamma{:}\mathrm{Type}.\,!\lambda_\,{:}!\mathcal{A}(\emptyset)\ \gamma.$
$\qquad \mathrm{let}\ !x_1 = \mathit{egetright}\ \mathrm{in}$
$\qquad \mathrm{let}\ !x_2 = \mathit{erefl}\ \mathrm{in}$
$\qquad !\mathrm{pack}[X\ \alpha,\ !\langle x_1, x_2\rangle]\,\mathrm{as}\,\mathcal{A}(\{A^+\})\ (\gamma \otimes X\ \alpha)$

$$\frac{\Delta \vdash A_1 = A_2 : \mathrm{Cap}^+}{\Delta \vdash \emptyset \boxplus A_1 \leq A_2 \rightsquigarrow \mathit{subaeq}}$$

$\mathit{subaeq} =$
$\qquad !\lambda\gamma{:}\mathrm{Type}.\,!\lambda\,!x_1{:}!\mathcal{A}(A_1)\ \gamma.$
$\qquad \mathrm{let}\ !x_2 = (\#1\,e\ \gamma)\ x_1\ \mathrm{in}$
$\qquad \mathrm{let}\ [\delta, \langle\, !x_{PQ},\ !x_A\rangle] = x_2\ \mathrm{in}$
$\qquad \mathrm{let}\ !y_1 = \mathit{egetleft}\ \mathrm{in}$
$\qquad \mathrm{let}\ !y_2 = \mathit{etrans}\ y_1\ x_{PQ}\ \mathrm{in}$
$\qquad !\mathrm{pack}[\delta,\ !\langle y_2, x_A\rangle]\,\mathrm{as}\,\mathcal{A}(A_2)\ (\gamma \otimes ())$

$$\frac{\Delta \vdash U_1 \boxplus A_1 \leq A_2 \qquad \Delta \vdash U_2 \boxplus A_2 \leq A_3}{\Delta \vdash U_1 \oplus U_2 \boxplus A_1 \leq A_3 \rightsquigarrow \mathit{subtrans}}$$

$\mathit{subtrans} =$
$\qquad !\lambda\gamma{:}\mathrm{Type}.\,!\lambda\,!x_1{:}!\mathcal{A}(A_1)\ \gamma.$
$\qquad \mathrm{let}\ !x_2 = e_1\ \gamma\ x_1\ \mathrm{in}$
$\qquad \mathrm{let}\ !x_3 = e_2\ \gamma \otimes u_1\ x_2\ \mathrm{in}$
$\qquad \mathrm{let}\ [\delta, \langle\, !z,\ !x_A\rangle] = x_3\ \mathrm{in}$
$\qquad \mathrm{let}\ !y = \mathit{eextractinter}\ \mathit{ilprodassoc}\ \mathrm{in}$
$\qquad \mathrm{let}\ !z' = \mathit{etrans}\ y\ z\ \mathrm{in}$
$\qquad !\mathrm{pack}[\delta,\ !\langle z', x_A\rangle]\,\mathrm{as}\,\mathcal{A}(A_3)\ (\gamma \otimes \mathcal{U}(U_1 \oplus U_2))$

$$\frac{\Delta \vdash U_1 \boxplus A_1 \leq A_1' \qquad \Delta \vdash U_2 \boxplus A_2 \leq A_2'}{\Delta \vdash U_1 \oplus U_2 \boxplus A_1 \oplus A_2 \leq A_1' \oplus A_2' \rightsquigarrow \mathit{subcong}}$$

$\mathit{subcong} =$
$\qquad !\lambda\gamma{:}\mathrm{Type}.\,!\lambda\,!x{:}!\mathcal{A}(A_1 \oplus A_2)\ \gamma.$
$\qquad \mathrm{let}\ [\delta, \langle\, !x_{PQ},\ !x_{A12}\rangle] = x\ \mathrm{in}$
$\qquad \mathrm{let}\ \langle\, !x_{A1},\ !x_{A2}\rangle = x_{A12}\ \mathrm{in}$
$\qquad \mathrm{let}\ !x_1' = e_1\ \delta\ x_{A1}\ \mathrm{in}$
$\qquad \mathrm{let}\ !x_2' = e_2\ \delta\ x_{A2}\ \mathrm{in}$
$\qquad \mathrm{let}\ [\delta_1, \langle\, !x_{Q1},\ !x_{A1}'\rangle] = x_1'\ \mathrm{in}$
$\qquad \mathrm{let}\ [q_2, \langle\, !x_{Q2},\ !x_{A2}'\rangle] = x_2'\ \mathrm{in}$

let $!x_{R1} = \mathit{egetrightleft}$ in
let $!x_{R2} = \mathit{egetrightright}$ in
let $!x_{S1} = \mathit{etrans}\ x_{R1}\ x_{Q1}$ in
let $!x_{S2} = \mathit{etrans}\ x_{R2}\ x_{Q2}$ in
let $!y_1 = !\mathrm{pack}[\delta_1,\ !\langle x_{S1}, x'_{A1}\rangle]\ \mathrm{as}\ \mathcal{A}(A'_1)\ (\delta \otimes \mathcal{U}(U_1 \oplus U_2))$ in
let $!y_2 = !\mathrm{pack}[q_2,\ !\langle x_{S2}, x'_{A2}\rangle]\ \mathrm{as}\ \mathcal{A}(A'_2)\ (\delta \otimes \mathcal{U}(U_1 \oplus U_2))$ in
let $!z = \mathit{elprodleft}\ x_{PQ}$ in
$!\mathrm{pack}[\delta \otimes \mathcal{U}(U_1 \oplus U_2),\ !\langle z,\ !\langle y_1, y_2\rangle\rangle]\ \mathrm{as}\ \mathcal{A}(A'_1 \oplus A'_2)\ (\gamma \otimes \mathcal{U}(U_1 \oplus U_2))$

$$\frac{\Delta \vdash c : \kappa}{\Delta \vdash c = c : \kappa \rightsquigarrow \mathit{irefl}}\ \kappa = \mathrm{Type}$$

$$\frac{\Delta \vdash c_2 = c_1 : \kappa \rightsquigarrow e}{\Delta \vdash c_1 = c_2 : \kappa \rightsquigarrow \mathit{isymm}\ e}\ \kappa = \mathrm{Type}$$

$$\frac{\Delta \vdash c_1 = c_2 : \kappa \rightsquigarrow e_1 \qquad \Delta \vdash c_2 = c_3 : \kappa \rightsquigarrow e_2}{\Delta \vdash c_1 = c_3 : \kappa \rightsquigarrow \mathit{itrans}\ e_1\ e_2}\ \kappa = \mathrm{Type}$$

$$\frac{\Delta \vdash c : \kappa}{\Delta \vdash c = c : \kappa \rightsquigarrow \mathit{irefl}}\ \kappa = \mathrm{Cap}^1$$

$$\frac{\Delta \vdash c_2 = c_1 : \kappa \rightsquigarrow e}{\Delta \vdash c_1 = c_2 : \kappa \rightsquigarrow \mathit{isymm}\ e}\ \kappa = \mathrm{Cap}^1$$

$$\frac{\Delta \vdash c_1 = c_2 : \kappa \rightsquigarrow e_1 \qquad \Delta \vdash c_2 = c_3 : \kappa \rightsquigarrow e_2}{\Delta \vdash c_1 = c_3 : \kappa \rightsquigarrow \mathit{itrans}\ e_1\ e_2}\ \kappa = \mathrm{Cap}^1$$

$$\frac{\Delta \vdash c : \kappa}{\Delta \vdash c = c : \kappa \rightsquigarrow \lambda\gamma{:}\mathrm{Type}.\mathit{irefl}}\ \kappa = \mathrm{Cap}^+$$

$$\frac{\Delta \vdash c_2 = c_1 : \kappa \rightsquigarrow e}{\Delta \vdash c_1 = c_2 : \kappa \rightsquigarrow \lambda\gamma{:}\mathrm{Type}.\mathit{isymm}\ (e\ \gamma)}\ \kappa = \mathrm{Cap}^+$$

$$\frac{\Delta \vdash c_1 = c_2 : \kappa \rightsquigarrow e_1 \qquad \Delta \vdash c_2 = c_3 : \kappa \rightsquigarrow e_2}{\Delta \vdash c_1 = c_3 : \kappa \rightsquigarrow \lambda\gamma{:}\mathrm{Type}.\mathit{itrans}\ (e_1\ \gamma)\ (e_2\ \gamma)}\ \kappa = \mathrm{Cap}^+$$

$$\frac{\Delta \vdash \tau_1 = \tau'_1 : \mathrm{Type} \rightsquigarrow e_1 \qquad \Delta \vdash \tau_2 = \tau'_2 : \mathrm{Type} \rightsquigarrow e_2}{\Delta \vdash \tau_1 \times \tau_2 = \tau'_1 \times \tau'_2 : \mathrm{Type} \rightsquigarrow \mathit{iprod}\ e_1\ e_2}$$

$$\frac{\Delta, \alpha : \kappa \vdash \tau = \tau' : \mathrm{Type} \rightsquigarrow e}{\Delta \vdash \forall\alpha{:}\kappa.\tau = \forall\alpha{:}\kappa.\tau' : \mathrm{Type} \rightsquigarrow \mathit{iall}\ (!(\lambda\alpha{:}\mathcal{K}(\kappa).e))}\ (\alpha \notin \mathit{domain}(\Delta))$$

$$\frac{\begin{array}{ccc}\Delta \vdash U_1 = U'_1 : \mathrm{Cap}^1 \rightsquigarrow e_{U1} & \ldots & \Delta \vdash U_n = U'_n : \mathrm{Cap}^1 \rightsquigarrow e_{Un} \\ \Delta \vdash A_1 = A'_1 : \mathrm{Cap}^+ \rightsquigarrow e_{A1} & \ldots & \Delta \vdash A_n = A'_n : \mathrm{Cap}^+ \rightsquigarrow e_{An} \\ \multicolumn{3}{c}{\Delta, \alpha{:}\mathrm{Cap}^+\ \mathrm{where}\ U_1 \boxplus \alpha \leq A_1, \ldots, U_n \boxplus \alpha \leq A_n \vdash \tau = \tau' : \mathrm{Type} \rightsquigarrow e}\end{array}}{\begin{array}{c}\Delta \vdash \forall\alpha{:}\mathrm{Cap}^+\ \mathrm{where}\ U_1 \boxplus \alpha \leq A_1, \ldots, U_n \boxplus \alpha \leq A_n.\tau \\ = \forall\alpha{:}\mathrm{Cap}^+\ \mathrm{where}\ U'_1 \boxplus \alpha \leq A'_1, \ldots, U'_n \boxplus \alpha \leq A'_n.\tau' : \mathrm{Type} \rightsquigarrow \mathit{teqbound}\end{array}}\ (\alpha \notin \mathit{domain}(\Delta))$$

$\mathit{teqbound} = \mathit{iall}\ (!(\lambda\alpha{:}\mathrm{Type} \to \mathrm{Type}.\mathit{ifun}\ s_1\ (\ldots(\mathit{ifun}\ s_n\ e)\ldots)))$
$s_k = (\mathit{iall}\ (!(\lambda\gamma{:}\mathrm{Type}.\mathit{ifunright}\ (\mathit{itrans}\ (e_{Ak}\ \gamma \otimes u_1)$
$(\mathit{iexist}\ (!(\lambda\delta{:}\mathrm{Type}.\mathit{iprodleft}\ (\mathit{iextractleft}\ (\mathit{ilprodright}\ e_{Uk})))))))))$

$$\frac{\Delta \vdash \tau = \tau' : \mathrm{Type} \rightsquigarrow e_\tau \qquad \Delta \vdash U = U' : \mathrm{Cap}^1 \rightsquigarrow e_U \qquad \Delta \vdash A = A' : \mathrm{Cap}^+ \rightsquigarrow e_A}{\Delta \vdash (C, \tau) \to 0 = (C', \tau') \to 0 : \mathrm{Type} \rightsquigarrow \mathit{teqfun}}$$

$\mathit{teqfun} = \mathit{iall}\ (!(\lambda\gamma{:}\mathrm{Type}.\mathit{ifunright}\ (\mathit{ilfun}\ e_U\ (\mathit{ilfun}\ (e_A\ \gamma)\ (\mathit{ilfunleft}\ e_\tau)))))$

C.4.1 Lemmas for CC2→linear $F\omega$

- If $\vdash \Delta$ and $\Delta \vdash U : \text{Cap}^1$ then $\mathbf{\Delta}(\Delta), \chi : \text{Type} \to \text{Type} \vdash \mathcal{U}(U) : \text{Type}$
- If $\vdash \Delta$ and $\Delta \vdash A : \text{Cap}^+$ then $\mathbf{\Delta}(\Delta), \chi : \text{Type} \to \text{Type} \vdash \mathcal{A}(A) : \text{Type} \to \text{Type}$
- If $\vdash \Delta$ and $\Delta \vdash \tau : \text{Type}$ then $\mathbf{\Delta}(\Delta), \chi : \text{Type} \to \text{Type} \vdash \mathcal{T}(\tau) : \text{Type}$
- If $\vdash \Delta$ and $\Delta \vdash U_1 = U_2 : \text{Cap}^1 \rightsquigarrow e$ then $\mathbf{\Delta}(\Delta), \chi : \text{Type} \to \text{Type}, \mathbf{\Gamma}(\Delta; \cdot) \vdash e : \mathcal{U}(U_1) \leftrightarrow \mathcal{U}(U_2)$
- If $\vdash \Delta$ and $\Delta \vdash A_1 = A_2 : \text{Cap}^+ \rightsquigarrow e$ then $\mathbf{\Delta}(\Delta), \chi : \text{Type} \to \text{Type}, \mathbf{\Gamma}(\Delta; \cdot) \vdash e : !\forall\gamma : \text{Type}. !(\mathcal{A}(A_1)\ \gamma) \leftrightarrow !(\mathcal{A}(A_2)\ \gamma)$
- If $\vdash \Delta$ and $\Delta \vdash U_1 \boxplus A_1 \leq A_2 \rightsquigarrow e$ then $\mathbf{\Delta}(\Delta), \chi : \text{Type} \to \text{Type}, \mathbf{\Gamma}(\Delta; \cdot) \vdash e : !\mathcal{S}(U_1 \boxplus A_1 \leq A_2)$
- If $\vdash \Delta$ and $\Delta \vdash \tau_1 = \tau_2 : \text{Type} \rightsquigarrow e$ then $\mathbf{\Delta}(\Delta), \chi : \text{Type} \to \text{Type}, \mathbf{\Gamma}(\Delta; \cdot) \vdash e : !\mathcal{T}(\tau_1) \leftrightarrow !\mathcal{T}(\tau_2)$
- If $\vdash \Delta$ and $\Delta \vdash \alpha' : \text{Cap}^+$ and $\Delta \vdash A : \text{Cap}^+$ then $\mathcal{A}([\alpha' \leftarrow A']A) = [\alpha' \leftarrow \mathcal{A}[A']]\mathcal{A}(A)$
- If $\vdash \Delta$ and $\Delta \vdash \alpha' : \text{Cap}^+$ and $\Delta \vdash U : \text{Cap}^1$ then $\mathcal{U}([\alpha' \leftarrow A']U) = [\alpha' \leftarrow A'']\mathcal{U}(U)$ for any A''
- If $\vdash \Delta$ and $\Delta \vdash \alpha' : \text{Cap}^1$ and $\Delta \vdash A : \text{Cap}^+$ then $\mathcal{A}([\alpha' \leftarrow U']A) = [\alpha' \leftarrow U'']\mathcal{A}(A)$ for any U''
- If $\vdash \Delta$ and $\Delta \vdash \alpha' : \text{Cap}^1$ and $\Delta \vdash U : \text{Cap}^1$ then $\mathcal{U}([\alpha' \leftarrow U']U) = [\alpha' \leftarrow \mathcal{U}(U')]\mathcal{U}(U)$
- If $\vdash \Delta$ and $\Delta \vdash \alpha' : \text{Cap}^+$ and $\Delta \vdash \tau : \text{Type}$ then $\mathcal{T}([\alpha' \leftarrow A']\tau) = [\alpha' \leftarrow \mathcal{A}[A']]\mathcal{T}(\tau)$
- If $\vdash \Delta$ and $\Delta \vdash \alpha' : \text{Cap}^1$ and $\Delta \vdash \tau : \text{Type}$ then $\mathcal{T}([\alpha' \leftarrow U']\tau) = [\alpha' \leftarrow \mathcal{U}(U')]\mathcal{T}(\tau)$
- If $\vdash \Delta$ and $\Delta \vdash \alpha' : \text{Type}$ and $\Delta \vdash \tau : \text{Type}$ then $\mathcal{T}([\alpha' \leftarrow \tau']\tau) = [\alpha' \leftarrow \mathcal{T}(\tau')]\mathcal{T}(\tau)$
- If $\vdash \Delta$ and $\Delta \vdash \Gamma$ then:
  - If $\Delta; \Gamma \vdash v : \tau$ then $\mathbf{\Delta}(\Delta), \chi : \text{Type} \to \text{Type}; \mathbf{\Gamma}(\Delta; \Gamma), \Gamma' \vdash \mathcal{V}(v) : !\mathcal{T}(\tau)$
  - If $\Delta; \Gamma \vdash h : \tau$ then $\mathbf{\Delta}(\Delta), \chi : \text{Type} \to \text{Type}; \mathbf{\Gamma}(\Delta; \Gamma), \Gamma' \vdash \mathcal{H}(h) : !\mathcal{T}(\tau)$
  - If $\Delta; \Gamma; U \boxplus A \vdash e$ then $\mathbf{\Delta}(\Delta), \chi : \text{Type} \to \text{Type}, \gamma : \text{Type}; \mathbf{\Gamma}(\Delta; \Gamma), \Gamma', \Gamma'' \vdash \mathcal{E}(e) : \text{true}$

where $\Gamma' = !(new : () \multimap \exists\rho : \text{Type}.\chi\ \rho), !(free : \forall\rho : \text{Type}.\chi\ \rho \to ()), !(use : \forall\rho : \text{Type}.\chi\ \rho \to \chi\ \rho)$

where $\Gamma'' = z_P : \gamma, z_U : \mathcal{U}(U), !(z_A : \mathcal{A}(A)\ \gamma)$

D Translating CC/CCL to CC/SLL

This section applies the techniques from the CC0-to-LC translation to the original calculus of capabilities [7] (referred to here as "CC/CCL"). Sections D.1 and D.2 define CC/CCL in two pieces: a small logic inside the type system, called CCL (section D.2), and the logic-independent part of the language (section D.1). Section D.3 defines an alternate logic, SLL, based on LC, and adapts the soundness proof of Walker, Crary, and Morrisett [8] to show the soundness of CC/SLL. Section D.4 translates CC/CCL to CC/SLL and describes the proof of the translation's type correctness.

D.1 Calculus of capabilities (logic-independent portion)

The syntax and rules are taken directly from [7] and [8].

$$
\begin{array}{rrcl}
\textit{kinds} & \kappa & = & \mathrm{Type} \mid \mathrm{Res} \mid \mathrm{Cap} \\
\textit{constructors} & c & = & \alpha \mid \tau \mid r \mid C \\
\textit{ctor vars}\ \alpha, \beta, \epsilon, \rho, \ldots & & & \\
\textit{types} & \tau & = & \alpha \mid \mathrm{int} \mid r\,\mathrm{handle} \mid \forall[\Delta](C, \tau_1, \ldots, \tau_n) \to 0\,\mathrm{at}\,r \mid \langle \tau_1, \ldots, \tau_n \rangle\,\mathrm{at}\,r \\
\textit{regions} & r & = & \rho \mid \nu \\
\\
\textit{ctor ctxts} & \Delta & = & \cdot \mid \Delta, \alpha : \kappa \\
\textit{value ctxts} & \Gamma & = & \cdot \mid \Gamma, x : \tau \\
\textit{region types} & \Upsilon & = & \{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\} \\
\textit{memory types} & \Psi & = & \{\nu_1 : \Upsilon_1, \ldots, \nu_n : \Upsilon_n\} \\
\\
\textit{word values} & v & = & x \mid i \mid \nu.\ell \mid \mathrm{handle}(\nu) \mid v[c] \\
\textit{heap values} & h & = & \mathrm{fix}\, f[\Delta](C, x_1 : \tau_1, \ldots x_n : \tau_n).e \mid \langle v_1, \ldots, v_n \rangle \\
\textit{arithmetic ops} & p & = & + \mid - \mid \times \\
\textit{declarations} & d & = & x = v \mid x = v_1\, p\, v_2 \mid x = h\,\mathrm{at}\,v \mid x = \#n\, v \mid \mathrm{newrgn}\,\rho, x \mid \mathrm{freergn}\,v \\
\textit{expressions} & e & = & \mathrm{let}\,d\,\mathrm{in}\,e \mid \mathrm{if0}\,v\,\mathrm{then}\,e_2\,\mathrm{else}\,e_3 \mid v(v_1, \ldots, v_n) \mid \mathrm{halt}\,v \\
\\
\textit{memory regions} & R & = & \{\ell_1 \mapsto h_1, \ldots, \ell_n \mapsto h_n\} \\
\textit{memories} & M & = & \{\nu_1 \mapsto R_1, \ldots, \nu_n \mapsto R_n\} \\
\textit{machine states} & P & = & (M, e)
\end{array}
$$

$$
\Delta \vdash \cdot \qquad \frac{\Delta \vdash \Delta'}{\Delta \vdash \Delta', \alpha : \kappa}(\alpha \notin \mathit{domain}(\Delta, \Delta'))
$$

$$
\Delta \vdash \cdot \qquad \frac{\Delta \vdash \Gamma \qquad \Delta \vdash \tau : \mathrm{Type}}{\Delta \vdash \Gamma, x : \tau}(x \notin \mathit{domain}(\Gamma))
$$

$$
\ldots, \alpha : \kappa, \ldots \vdash \alpha : \kappa \qquad \Delta \vdash \mathrm{int} : \mathrm{Type} \qquad \frac{\Delta \vdash r : \mathrm{Res}}{\Delta \vdash r\,\mathrm{handle} : \mathrm{Type}}
$$

$$
\Delta \vdash \nu : \mathrm{Res} \qquad \frac{\Delta \vdash \tau_1 : \mathrm{Type} \qquad \ldots \qquad \Delta \vdash \tau_n : \mathrm{Type} \qquad \Delta \vdash r : \mathrm{Res}}{\Delta \vdash \langle \tau_1, \ldots, \tau_n \rangle\,\mathrm{at}\,r : \mathrm{Type}}
$$

$$
\frac{\Delta \vdash \Delta' \qquad \Delta, \Delta' \vdash \tau_1 : \mathrm{Type} \ldots \Delta, \Delta' \vdash \tau_n : \mathrm{Type} \qquad \Delta, \Delta' \vdash C : \mathrm{Cap} \qquad \Delta \vdash r : \mathrm{Res}}{\Delta \vdash \forall[\Delta'](C, \tau_1, \ldots, \tau_n) \to 0\,\mathrm{at}\,r : \mathrm{Type}}
$$

$$
\frac{\vdash \Upsilon_1 \qquad \ldots \qquad \vdash \Upsilon_n}{\vdash \{\nu_1 : \Upsilon_1, \ldots, \nu_n : \Upsilon_n\}} \qquad \frac{\cdot \vdash \tau_1 : \mathrm{Type} \qquad \ldots \qquad \cdot \vdash \tau_n : \mathrm{Type}}{\vdash \{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\}}
$$

$$
\Delta \vdash \cdot = \cdot \qquad \frac{\Delta \vdash \Delta_1 = \Delta_2}{\Delta \vdash \Delta_1, \alpha : \kappa = \Delta_2, \alpha : \kappa}(\alpha \notin \mathit{domain}(\Delta, \Delta_1))
$$

$$\frac{\Delta \vdash c : \kappa}{\Delta \vdash c = c : \kappa} \qquad \frac{\Delta \vdash c_2 = c_1 : \kappa}{\Delta \vdash c_1 = c_2 : \kappa} \qquad \frac{\Delta \vdash c_1 = c_2 : \kappa \qquad \Delta \vdash c_2 = c_3 : \kappa}{\Delta \vdash c_1 = c_3 : \kappa}$$

$$\frac{\Delta \vdash \tau_1 = \tau_1' : \mathrm{Type} \qquad \ldots \qquad \Delta \vdash \tau_n = \tau_n' : \mathrm{Type}}{\Delta \vdash \langle \tau_1, \ldots, \tau_n \rangle \,\mathrm{at}\, r = \langle \tau_1', \ldots, \tau_n' \rangle \,\mathrm{at}\, r : \mathrm{Type}}$$

$$\frac{\begin{array}{c}\Delta'' \vdash \Delta = \Delta' \qquad \Delta'', \Delta \vdash C = C' : \mathrm{Cap} \\ \Delta'', \Delta \vdash \tau_1 = \tau_1' : \mathrm{Type} \qquad \ldots \qquad \Delta'', \Delta \vdash \tau_n = \tau_n' : \mathrm{Type}\end{array}}{\Delta'' \vdash \forall[\Delta](C, \tau_1, \ldots, \tau_n) \to 0 \,\mathrm{at}\, r = \forall[\Delta'](C', \tau_1', \ldots, \tau_n') \to 0 \,\mathrm{at}\, r : \mathrm{Type}}$$

$$\frac{\begin{array}{c}\Delta \vdash \Delta' \qquad \Delta, \Delta' \vdash C : \mathrm{Cap} \\ \Delta, \Delta' \vdash \tau_1 : \mathrm{Type} \qquad \ldots \qquad \Delta, \Delta' \vdash \tau_n : \mathrm{Type} \qquad \Delta \vdash r : \mathrm{Res} \\ \Psi; \Delta, \Delta'; \Gamma, f : \tau_f, x_1 : \tau_1, \ldots, x_n : \tau_n; C \vdash e \\ \tau_f = \forall[\Delta'](C, \tau_1, \ldots, \tau_n) \to 0 \,\mathrm{at}\, r\end{array}}{\Psi; \Delta; \Gamma \vdash \mathrm{fix}\, f[\Delta'](C, x_1 : \tau_1, \ldots, x_n : \tau_n).e \,\mathrm{at}\, r : \tau_f} (f, x_1, \ldots, x_n \notin \mathit{domain}(\Gamma))$$

$$\frac{\Psi; \Delta; \Gamma \vdash v_1 : \tau_1 \qquad \ldots \qquad \Psi; \Delta; \Gamma \vdash v_n : \tau_n \qquad \Delta \vdash r : \mathrm{Res}}{\Psi; \Delta; \Gamma \vdash \langle v_1, \ldots, v_n \rangle \,\mathrm{at}\, r : \langle \tau_1, \ldots, \tau_n \rangle \,\mathrm{at}\, r}$$

$$\frac{\Psi; \Delta; \Gamma \vdash h \,\mathrm{at}\, r : \tau' \qquad \Delta \vdash \tau' = \tau : \mathrm{Type}}{\Psi; \Delta; \Gamma \vdash h \,\mathrm{at}\, r : \tau}$$

$$\Psi; \Delta; \ldots, x : \tau, \ldots \vdash x : \tau$$

$$\Psi; \Delta; \Gamma \vdash i : \mathrm{int}$$

$$\frac{\Delta \vdash \langle \tau_1, \ldots, \tau_n \rangle \,\mathrm{at}\, \nu : \mathrm{Type}}{\Psi; \Delta; \Gamma \vdash \nu.\ell : \langle \tau_1, \ldots, \tau_n \rangle \,\mathrm{at}\, \nu} (\nu \notin \mathit{domain}(\Psi))$$

$$\frac{\Delta \vdash \forall[\Delta'](C, \tau_1, \ldots, \tau_n) \to 0 \,\mathrm{at}\, \nu : \mathrm{Type}}{\Psi; \Delta; \Gamma \vdash \nu.\ell : \forall[\Delta'](C, \tau_1, \ldots, \tau_n) \to 0 \,\mathrm{at}\, \nu} (\nu \notin \mathit{domain}(\Psi))$$

$$\Psi; \Delta; \Gamma \vdash \nu.\ell : \Psi(\nu.\ell)$$

$$\Psi; \Delta; \Gamma \vdash \mathrm{handle}(\nu) : \nu \,\mathrm{handle}$$

$$\frac{\Psi; \Delta; \Gamma \vdash v : \forall[\alpha : \kappa, \Delta'](C, \tau_1, \ldots, \tau_n) \to 0 \,\mathrm{at}\, r \qquad \Delta \vdash c : \kappa}{\Psi; \Delta; \Gamma \vdash v[c] : [\alpha \leftarrow c](\forall[\Delta'](C, \tau_1, \ldots, \tau_n) \to 0 \,\mathrm{at}\, r)}$$

$$\frac{\Psi; \Delta; \Gamma \vdash v : \tau' \qquad \Delta \vdash \tau' = \tau : \mathrm{Type}}{\Psi; \Delta; \Gamma \vdash v : \tau}$$

$$\frac{\Psi; \Delta; \Gamma \vdash v : \tau}{\Psi; \Delta; \Gamma; C \vdash x = v \Longrightarrow \Delta; \Gamma, x : \tau; C} (x \notin \mathit{domain}(\Gamma))$$

$$\frac{\Psi;\Delta;\Gamma\vdash v_1:\mathrm{int}\qquad \Psi;\Delta;\Gamma\vdash v_2:\mathrm{int}}{\Psi;\Delta;\Gamma;C\vdash x=v_1\,p\,v_2\Longrightarrow\Delta;\Gamma,x:\mathrm{int};C}(x\notin domain(\Gamma))$$

$$\frac{\Psi;\Delta;\Gamma;C\vdash d\Longrightarrow\Delta';\Gamma';C'\qquad \Psi;\Delta';\Gamma';C'\vdash e}{\Psi;\Delta;\Gamma;C\vdash \mathrm{let}\,d\ \mathrm{in}\,e}$$

$$\frac{\Psi;\Delta;\Gamma\vdash v:\mathrm{int}\qquad \Psi;\Delta;\Gamma;C\vdash e_2\qquad \Psi;\Delta;\Gamma;C\vdash e_3}{\Psi;\Delta;\Gamma;C\vdash \mathrm{if0}\,v\,\mathrm{then}\,e_2\,\mathrm{else}\,e_3}$$

$$\frac{\vdash\Psi\qquad \Psi\vdash R_1\,\mathrm{at}\,\nu_1:\Upsilon_1\qquad\ldots\qquad \Psi\vdash R_n\,\mathrm{at}\,\nu_n:\Upsilon_n\qquad \Psi=\{\nu_1:\Upsilon_1,\ldots,\nu_n:\Upsilon_n\}}{\vdash\{\nu_1\mapsto R_1,\ldots,\nu_n\mapsto R_n\}:\Psi}$$

$$\frac{\Psi;\cdot;\cdot\vdash h_1\,\mathrm{at}\,\nu:\tau_1\qquad\ldots\qquad \Psi;\cdot;\cdot\vdash h_n\,\mathrm{at}\,\nu:\tau_n}{\Psi\vdash\{\ell_1\mapsto h_1,\ldots,\ell_n\mapsto h_n\}\,\mathrm{at}\,\nu:\{\ell_1:\tau_1,\ldots,\ell_n:\tau_n\}}$$

$$\frac{\vdash M:\Psi\qquad \Psi\vdash C\,sat\qquad \Psi;\cdot;\cdot;C\vdash e}{\vdash(M,e)}$$

D.2 CC/CCL

CC/CCL extends section D.1 with the following. The syntax and rules are taken directly from [7] and [8]. Note that [7] includes a rule $\Delta\vdash\overline{\{r^+\}}=\{r^+\}:\mathrm{Cap}$, while the technical report [8] omits this rule. This section omits the rule as well, since the rule can be derived from $\Delta\vdash\overline{\overline{C}}=\overline{C}:\mathrm{Cap}$ and $\Delta\vdash\overline{\{r^1\}}=\{r^+\}:\mathrm{Cap}$ and the congruence rule for $\overline{C}$:

$$\frac{\dfrac{\Delta\vdash\{r^+\}=\overline{\{r^1\}}:\mathrm{Cap}}{\Delta\vdash\overline{\{r^+\}}=\overline{\overline{\{r^1\}}}:\mathrm{Cap}}\qquad \dfrac{\Delta\vdash\overline{\{r^1\}}:\mathrm{Cap}}{\Delta\vdash\overline{\overline{\{r^1\}}}=\overline{\{r^1\}}:\mathrm{Cap}}\qquad \Delta\vdash\overline{\{r^1\}}=\{r^+\}:\mathrm{Cap}}{\Delta\vdash\overline{\{r^+\}}=\{r^+\}:\mathrm{Cap}}$$

$$\begin{array}{rcl}
\textit{capabilities} & C & = & \epsilon\mid\emptyset\mid\{r^{\varphi}\}\mid C_1\oplus C_2\mid\overline{C}\\
\textit{multiplicities} & \varphi & = & 1\mid +\\
\textit{ctor ctxts} & \Delta & = & \ldots\mid\Delta,\epsilon\le C
\end{array}$$

$$\frac{\Delta\vdash\Delta'\qquad \Delta,\Delta'\vdash C:\mathrm{Cap}}{\Delta\vdash\Delta',\alpha\le C}(\alpha\notin domain(\Delta,\Delta'))$$

$$\ldots,\alpha\le C,\ldots\vdash\alpha:\mathrm{Cap}\qquad\qquad \Delta\vdash\emptyset:\mathrm{Cap}\qquad\qquad \frac{\Delta\vdash r:\mathrm{Res}}{\Delta\vdash\{r^{\varphi}\}:\mathrm{Cap}}$$

$$\frac{\Delta\vdash C_1:\mathrm{Cap}\qquad \Delta\vdash C_2:\mathrm{Cap}}{\Delta\vdash C_1\oplus C_2:\mathrm{Cap}}\qquad\qquad \frac{\Delta\vdash C:\mathrm{Cap}}{\Delta\vdash\overline{C}:\mathrm{Cap}}$$

$$\frac{\Delta\vdash\Delta_1=\Delta_2\qquad \Delta,\Delta_1\vdash C_1=C_2:\mathrm{Cap}}{\Delta\vdash\Delta_1,\alpha\le C_1=\Delta_2,\alpha\le C_2}(\alpha\notin domain(\Delta,\Delta_1))$$

$$\frac{\Delta \vdash C_1 = C_1' : \mathrm{Cap} \qquad \Delta \vdash C_2 = C_2' : \mathrm{Cap}}{\Delta \vdash C_1 \oplus C_2 = C_1' \oplus C_2' : \mathrm{Cap}}$$

$$\frac{\Delta \vdash C : \mathrm{Cap}}{\Delta \vdash \emptyset \oplus C = C : \mathrm{Cap}} \qquad \frac{\Delta \vdash C_1 : \mathrm{Cap} \qquad \Delta \vdash C_2 : \mathrm{Cap}}{\Delta \vdash C_1 \oplus C_2 = C_2 \oplus C_1 : \mathrm{Cap}}$$

$$\frac{\Delta \vdash C_1 : \mathrm{Cap} \qquad \Delta \vdash C_2 : \mathrm{Cap} \qquad \Delta \vdash C_3 : \mathrm{Cap}}{\Delta \vdash (C_1 \oplus C_2) \oplus C_3 = C_1 \oplus (C_2 \oplus C_3) : \mathrm{Cap}}$$

$$\frac{\Delta \vdash C = C' : \mathrm{Cap}}{\Delta \vdash \overline{C} = \overline{C'} : \mathrm{Cap}} \qquad \frac{\Delta \vdash C : \mathrm{Cap}}{\Delta \vdash \overline{C} = \overline{C} \oplus \overline{C} : \mathrm{Cap}}$$

$$\Delta \vdash \overline{\emptyset} = \emptyset : \mathrm{Cap} \qquad \frac{\Delta \vdash r : \mathrm{Res}}{\Delta \vdash \overline{\{r^1\}} = \{r^+\} : \mathrm{Cap}}$$

$$\frac{\Delta \vdash C : \mathrm{Cap}}{\Delta \vdash \overline{\overline{C}} = \overline{C} : \mathrm{Cap}} \qquad \frac{\Delta \vdash C_1 : \mathrm{Cap} \qquad \Delta \vdash C_2 : \mathrm{Cap}}{\Delta \vdash \overline{C_1 \oplus C_2} = \overline{C_1} \oplus \overline{C_2} : \mathrm{Cap}}$$

$$\frac{\Delta \vdash C_1 = C_2 : \mathrm{Cap}}{\Delta \vdash C_1 \leq C_2} \qquad \frac{\Delta \vdash C_1 \leq C_2 \qquad \Delta \vdash C_2 \leq C_3}{\Delta \vdash C_1 \leq C_3}$$

$$\frac{\Delta \vdash C_1 \leq C_1' \qquad \Delta \vdash C_2 \leq C_2'}{\Delta \vdash C_1 \oplus C_2 \leq C_1' \oplus C_2'}$$

$$\frac{\Delta \vdash C \leq C'}{\Delta \vdash \overline{C} \leq \overline{C'}} \qquad \ldots, \alpha \leq C, \ldots \vdash \alpha \leq C \qquad \frac{\Delta \vdash C : \mathrm{Cap}}{\Delta \vdash C \leq \overline{C}}$$

$$\frac{\Psi; \Delta; \Gamma \vdash v : \forall[\alpha \leq C'', \Delta'](C', \tau_1, \ldots, \tau_n) \rightarrow 0 \,\mathrm{at}\, r \qquad \Delta \vdash C \leq C''}{\Psi; \Delta; \Gamma \vdash v[C] : [\alpha \leftarrow C](\forall[\Delta'](C', \tau_1, \ldots, \tau_n) \rightarrow 0 \,\mathrm{at}\, r)}$$

$$\frac{\Psi; \Delta; \Gamma \vdash v : r \,\mathrm{handle} \qquad \Psi; \Delta; \Gamma \vdash h \,\mathrm{at}\, r : \tau \qquad \Delta \vdash C \leq C' \oplus \{r^+\}}{\Psi; \Delta; \Gamma; C \vdash x = h \,\mathrm{at}\, v \Longrightarrow \Delta; \Gamma, x : \tau; C} (x \notin \mathit{domain}(\Gamma))$$

$$\frac{\Psi; \Delta; \Gamma \vdash v : \langle \tau_0, \ldots \tau_{n-1} \rangle \,\mathrm{at}\, r \qquad \Delta \vdash C \leq C' \oplus \{r^+\}}{\Psi; \Delta; \Gamma; C \vdash x = \#k\, v \Longrightarrow \Delta; \Gamma, x : \tau_k; C} (x \notin \mathit{domain}(\Gamma) \,\mathit{and}\, 0 \leq k < n)$$

$$\Psi; \Delta; \Gamma; C \vdash \mathrm{newrgn}\, \alpha, x \Longrightarrow \Delta, \alpha : \mathrm{Res}; \Gamma, x : \alpha \,\mathrm{handle}; C \oplus \{\alpha^1\}$$
$$(\alpha \notin \mathit{domain}(\Delta) \,\mathit{and}\, x \notin \mathit{domain}(\Gamma))$$

$$\frac{\Psi; \Delta; \Gamma \vdash v : r \,\mathrm{handle} \qquad \Delta \vdash C = C' \oplus \{r^1\} : \mathrm{Cap}}{\Psi; \Delta; \Gamma; C \vdash \mathrm{freergn}\, v \Longrightarrow \Delta; \Gamma; C'}$$

$$\frac{\begin{array}{c} \Psi; \Delta; \Gamma \vdash v : \forall[](C', \tau_1, \ldots, \tau_n) \rightarrow 0 \,\mathrm{at}\, r \\ \Psi; \Delta; \Gamma \vdash v_1 : \tau_1 \qquad \ldots \qquad \Psi; \Delta; \Gamma \vdash v_n : \tau_n \\ \Delta \vdash C \leq C'' \oplus \{r^+\} \qquad \Delta \vdash C \leq C' \end{array}}{\Psi; \Delta; \Gamma; C \vdash v(v_1, \ldots, v_n)}$$

$$\frac{\Psi; \Delta; \Gamma \vdash v : \mathrm{int} \qquad \Delta \vdash C = \emptyset : \mathrm{Cap}}{\Psi; \Delta; \Gamma; C \vdash \mathrm{halt}\, v}$$

$$\frac{\cdot \vdash C = \{\nu_1^{\varphi_1}\} \oplus \ldots \oplus \{\nu_n^{\varphi_n}\} : \mathrm{Cap}}{\{\nu_1 : \Upsilon_1, \ldots, \nu_n : \Upsilon_n\} \vdash C \,\mathit{sat}} (\mathit{all}\, \nu_i \,\mathit{distinct})$$

D.3 CC/SLL

CC/SLL extends section D.1 with the following.

$$\begin{array}{rrcl} \textit{capabilities} & C & = & \ldots \mid \{\rho\} \mid C_1 \otimes C_2 \mid C_1 \& C_2 \mid C_1 \multimap C_2 \mid \mathrm{true} \\ \textit{cap ctxts} & \Lambda & = & C_1, \ldots, C_n \end{array}$$

$$\Delta \vdash \emptyset : \mathrm{Cap} \qquad \frac{\Delta \vdash r : \mathrm{Res}}{\Delta \vdash \{r\} : \mathrm{Cap}} \qquad \Delta \vdash \mathrm{true} : \mathrm{Cap}$$

$$\frac{\Delta \vdash C_1 : \mathrm{Cap} \qquad \Delta \vdash C_2 : \mathrm{Cap}}{\Delta \vdash C_1 \otimes C_2 : \mathrm{Cap}} \qquad \frac{\Delta \vdash C_1 : \mathrm{Cap} \qquad \Delta \vdash C_2 : \mathrm{Cap}}{\Delta \vdash C_1 \& C_2 : \mathrm{Cap}}$$

$$\frac{\Delta \vdash C_1 : \mathrm{Cap} \qquad \Delta \vdash C_2 : \mathrm{Cap}}{\Delta \vdash C_1 \multimap C_2 : \mathrm{Cap}}$$

$$C \vdash C \qquad \vdash \emptyset \qquad \frac{\Lambda \vdash C}{\Lambda, \emptyset \vdash C} \qquad \Lambda \vdash \mathrm{true}$$

$$\frac{\Lambda_1 \vdash C_1 \qquad \Lambda_2 \vdash C_2}{\Lambda_1, \Lambda_2 \vdash C_1 \otimes C_2} \qquad \frac{\Lambda \vdash C_1 \qquad \Lambda \vdash C_2}{\Lambda \vdash C_1 \& C_2} \qquad \frac{\Lambda, C_1 \vdash C_2}{\Lambda \vdash C_1 \multimap C_2}$$

$$\frac{\Lambda, C_1, C_2 \vdash C_3}{\Lambda, C_1 \otimes C_2 \vdash C_3} \qquad \frac{\Lambda, C_k \vdash C_3}{\Lambda, C_1 \& C_2 \vdash C_3}(k \in \{1,2\}) \qquad \frac{\Lambda_1 \vdash C_1 \qquad \Lambda_2, C_2 \vdash C_3}{\Lambda_1, \Lambda_2, C_1 \multimap C_2 \vdash C_3}$$

$$\frac{\Delta \vdash C_1 : \kappa \qquad \Delta \vdash C_2 : \kappa \qquad C_1 \vdash C_2 \qquad C_2 \vdash C_1}{\Delta \vdash C_1 = C_2 : \kappa}$$

$$\frac{\Delta \vdash C_1 : \kappa \qquad \Delta \vdash C_2 : \kappa \qquad C_1 \vdash C_2}{\Delta \vdash C_1 \leq C_2}$$

$$\frac{\Psi; \Delta; \Gamma \vdash v : r\,\mathrm{handle} \qquad \Psi; \Delta; \Gamma \vdash h\,\mathrm{at}\,r : \tau \qquad \Delta \vdash C \leq \{r\} \otimes \mathrm{true}}{\Delta; \Gamma; C \vdash x = h\,\mathrm{at}\,v \Longrightarrow \Delta; \Gamma, x : \tau; C}(x \notin \mathit{domain}(\Gamma))$$

$$\frac{\Psi; \Delta; \Gamma \vdash v : \langle \tau_0, \ldots \tau_{n-1} \rangle\,\mathrm{at}\,r \qquad \Delta \vdash C \leq \{r\} \otimes \mathrm{true}}{\Psi; \Delta; \Gamma; C \vdash x = \#k\,v \Longrightarrow \Delta; \Gamma, x : \tau_k; C}(x \notin \mathit{domain}(\Gamma)\,\mathit{and}\,0 \leq k < n)$$

$$\frac{\Delta \vdash C \otimes \{\alpha\} = C' : \mathrm{Cap}}{\Psi; \Delta; \Gamma; C \vdash \mathrm{newrgn}\,\alpha, x \Longrightarrow \Delta, \alpha : \mathrm{Res}; \Gamma, x : \alpha\,\mathrm{handle}; C'}(\alpha \notin \mathit{domain}(\Delta)\,\mathit{and}\,x \notin \mathit{domain}(\Gamma))$$

$$\frac{\Psi; \Delta; \Gamma \vdash v : r\,\mathrm{handle} \qquad \Delta \vdash C = C' \otimes \{r\} : \mathrm{Cap}}{\Psi; \Delta; \Gamma; C \vdash \mathrm{freergn}\,v \Longrightarrow \Delta; \Gamma; C'}$$

$$\frac{\begin{array}{c} \Psi; \Delta; \Gamma \vdash v : \forall[](C', \tau_1, \ldots, \tau_n) \rightarrow 0\,\mathrm{at}\,r \\ \Psi; \Delta; \Gamma \vdash v_1 : \tau_1 \qquad \ldots \qquad \Psi; \Delta; \Gamma \vdash v_n : \tau_n \\ \Delta \vdash C \leq \{r\} \otimes \mathrm{true} \qquad \Delta \vdash C \leq C' \end{array}}{\Psi; \Delta; \Gamma; C \vdash v(v_1, \ldots, v_n)}$$

$$\frac{\Psi; \Delta; \Gamma \vdash v : \mathrm{int}}{\Psi; \Delta; \Gamma; C \vdash \mathrm{halt}\, v}$$

Define $\Psi = \Psi_1, \Psi_2$ iff $\Psi = \Psi_1 \cup \Psi_2$ and the domains of Ψ_1 and Ψ_2 are disjoint.

$$\{\} \vdash \emptyset\, sat$$

$$\{\nu : \Upsilon\} \vdash \{\nu\}\, sat$$

$$\frac{\Psi_1 \vdash C_1\, sat \qquad \Psi_2 \vdash C_2\, sat}{\Psi_1, \Psi_2 \vdash C_1 \otimes C_2\, sat}$$

$$\frac{\Psi \vdash C_1\, sat \qquad \Psi \vdash C_2\, sat}{\Psi \vdash C_1 \& C_2\, sat}$$

$$\frac{for\, all\, \Psi_1\, and\, \Psi_2.((\Psi_2 = \Psi, \Psi_1\, and\, \Psi_1 \vdash C_1\, sat)\, implies\, \Psi_2 \vdash C_2\, sat)}{\Psi \vdash C_1 \multimap C_2\, sat}$$

$$\Psi \vdash \mathrm{true}\, sat$$

$$\frac{\Psi_1 \vdash C_1\, sat \qquad \ldots \qquad \Psi_n \vdash C_n\, sat}{\Psi_1, \ldots, \Psi_n \vdash C_1, \ldots, C_n\, sat}$$

D.3.1 CC/SLL lemmas

- If $\Psi \vdash \Lambda\, sat$ and $\Lambda \vdash C$ then $\Psi \vdash C\, sat$. Proof: see [12].
- If $\vdash (M, e)$ and $(M, e) \longmapsto^* (M', e')$ then either $e' = \mathrm{halt}\, v$ or there is some (M'', e'') such that $(M', e') \longmapsto (M'', e'')$. Proof: based on proof in [8]; see [12].

D.4 Translation: CC/CCL→CC/SLL

From the CC0→CC1 translation, keep $\mathcal{C}()$, $\mathcal{S}()$, $\mathbf{\Delta}()$, changing $\{\rho^\varphi\}$ to $\{r^\varphi\}$. Call these $\mathcal{C}^1()$, $\mathcal{S}^1()$, $\mathbf{\Delta}^1()$.

From the CC1→CC2 translation, keep $\mathcal{U}()$, $\mathcal{A}()$, $\mathbf{\Delta}()$, $[]$, changing $\{\rho^\varphi\}$ to $\{r^\varphi\}$. Call these $\mathcal{U}^2()$, $\mathcal{A}^2()$, $\mathbf{\Delta}^2()$, $[]^2$.

From the CC2→LC translation, keep $\mathcal{U}()$, $\mathcal{A}()$, $\mathbf{\Delta}()$, $[]$, changing $\{\rho^\varphi\}$ to $\{r^\varphi\}$ and $\{\rho\}$ to $\{r\}$. Call these $\mathcal{U}^3()$, $\mathcal{A}^3()$, $\mathbf{\Delta}^3()$, $[]^3$.

$\mathcal{C}(\Delta, C) = [\Delta]\mathcal{C}(C)$
$\mathcal{C}(C) = \mathcal{U}^3(\mathcal{U}^2(\mathcal{C}^1(C))) \otimes \mathcal{A}^3(\mathcal{A}^2(\mathcal{C}^1(C))) \otimes \mathrm{true}$
$\mathbf{\Delta}(\Delta) = \mathbf{\Delta}^3(\mathbf{\Delta}^2(\mathbf{\Delta}^1(\Delta)))$
$[\Delta] = [\mathbf{\Delta}^2(\mathbf{\Delta}^1(\Delta))]^3 \mathcal{U}^3([\mathbf{\Delta}^1(\Delta)]^2)$
$\mathcal{U}^3([\alpha_1 \leftarrow U_1, \ldots, \alpha_n \leftarrow U_n]) = [\alpha_1 \leftarrow \mathcal{U}^3(U_1), \ldots, \alpha_n \leftarrow \mathcal{U}^3(U_n)]$
$\mathcal{T}(\alpha) = \alpha$
$\mathcal{T}(\mathrm{int}) = \mathrm{int}$
$\mathcal{T}(r\, \mathrm{handle}) = r\, \mathrm{handle}$
$\mathcal{T}(\forall[\Delta](C, \tau_1, \ldots, \tau_n) \rightarrow 0\, \mathrm{at}\, r) = \forall[\mathbf{\Delta}(\Delta)]([\Delta]\mathcal{C}(C), [\Delta]\mathcal{T}(\tau_1), \ldots, [\Delta]\mathcal{T}(\tau_n)) \rightarrow 0\, \mathrm{at}\, r$
$\mathcal{T}(\langle\tau_1, \ldots, \tau_n\rangle\, \mathrm{at}\, r) = \langle\mathcal{T}(\tau_1), \ldots, \mathcal{T}(\tau_n)\rangle\, \mathrm{at}\, r$

$\mathbf{\Gamma}(\cdot) = \cdot$
$\mathbf{\Gamma}(x : \tau, \Gamma) = x : \mathcal{T}(\tau), \mathbf{\Gamma}(\Gamma)$
$\mathbf{\Upsilon}(\{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\}) = \{\ell_1 : \mathcal{T}(\tau_1), \ldots, \ell_n : \mathcal{T}(\tau_n)\}$
$\mathbf{\Psi}(\{\nu_1 : \Upsilon_1, \ldots, \nu_n : \Upsilon_n\}) = \{\nu_1 : \mathbf{\Upsilon}(\Upsilon_1), \ldots, \nu_n : \mathbf{\Upsilon}(\Upsilon_n)\}$
The translations of v, h, d, and e are directed by typing judgments:

- $\Psi; \Delta; \Gamma \vdash v : \tau \rightsquigarrow \mathcal{V}(v)$
- $\Psi; \Delta; \Gamma \vdash h \,\mathrm{at}\, r : \tau \rightsquigarrow \mathcal{H}(h \,\mathrm{at}\, r)$
- $\Psi; \Delta; \Gamma; C \vdash d \Longrightarrow \Delta'; \Gamma'; C' \rightsquigarrow \mathcal{D}(d)$
- $\Psi; \Delta; \Gamma; C \vdash e \rightsquigarrow \mathcal{E}(e)$

For conciseness, though, most of the definitions below suppress the typing judgment when it is not immediately relevant.
$\mathcal{V}(x) = x$
$\mathcal{V}(i) = i$
$\mathcal{V}(\nu.\ell) = \nu.\ell$
$\mathcal{V}(\mathrm{handle}(\nu)) = \mathrm{handle}(\nu)$

$$\frac{\Psi; \Delta; \Gamma \vdash v : \forall[\alpha : \kappa, \Delta'](C, \tau_1, \ldots, \tau_n) \rightarrow 0 \,\mathrm{at}\, r \rightsquigarrow v' \qquad \Delta \vdash c : \kappa}{\Psi; \Delta; \Gamma \vdash v[c] : [\alpha \leftarrow c](\forall[\Delta'](C, \tau_1, \ldots, \tau_n) \rightarrow 0 \,\mathrm{at}\, r) \rightsquigarrow v'[c]} (\kappa = \mathrm{Res})$$

$$\frac{\Psi; \Delta; \Gamma \vdash v : \forall[\alpha : \kappa, \Delta'](C, \tau_1, \ldots, \tau_n) \rightarrow 0 \,\mathrm{at}\, r \rightsquigarrow v' \qquad \Delta \vdash c : \kappa}{\Psi; \Delta; \Gamma \vdash v[c] : [\alpha \leftarrow c](\forall[\Delta'](C, \tau_1, \ldots, \tau_n) \rightarrow 0 \,\mathrm{at}\, r) \rightsquigarrow v'[\mathcal{T}(c)]} (\kappa = \mathrm{Type})$$

$$\frac{\Psi; \Delta; \Gamma \vdash v : \forall[\alpha : \kappa, \Delta'](C, \tau_1, \ldots, \tau_n) \rightarrow 0 \,\mathrm{at}\, r \rightsquigarrow v' \qquad \Delta \vdash c : \kappa}{\begin{array}{c}\Psi; \Delta; \Gamma \vdash v[c] : [\alpha \leftarrow c](\forall[\Delta'](C, \tau_1, \ldots, \tau_n) \rightarrow 0 \,\mathrm{at}\, r) \\ \rightsquigarrow v'[\mathcal{A}^3(\mathcal{A}^2(\mathcal{S}^1(c)))][\mathcal{U}^3(\mathcal{U}^2(\mathcal{C}^1(c)))][\mathcal{A}^3(\mathcal{A}^2(\mathcal{C}^1(c)))]\end{array}} (\kappa = \mathrm{Cap})$$

$$\frac{\begin{array}{c}\Psi; \Delta; \Gamma \vdash v : \forall[\alpha \leq C'', \Delta'](C', \tau_1, \ldots, \tau_n) \rightarrow 0 \,\mathrm{at}\, r \rightsquigarrow v' \qquad \Delta \vdash C \leq C'' \\ \mathbf{\Delta}^1(\Delta) \vdash \mathcal{C}^1(C) \leq \mathcal{C}^1(C'') \rightsquigarrow U_B\end{array}}{\begin{array}{c}\Psi; \Delta; \Gamma \vdash v[C] : [\alpha \leftarrow C](\forall[\Delta'](C', \tau_1, \ldots, \tau_n) \rightarrow 0 \,\mathrm{at}\, r) \\ \rightsquigarrow v'[\mathcal{A}^3(\mathcal{A}^2(\mathcal{S}^1(C)))][\mathcal{U}^3(U_B)][\mathcal{A}^3(\mathcal{A}^2(\mathcal{C}^1(C)))]\end{array}}$$

$$\frac{\Psi; \Delta; \Gamma \vdash v : \tau' \rightsquigarrow v' \qquad \Delta \vdash \tau' = \tau : \mathrm{Type}}{\Psi; \Delta; \Gamma \vdash v : \tau \rightsquigarrow v'}$$

$$\frac{\Psi; \Delta; \Gamma \vdash h \,\mathrm{at}\, r : \tau' \rightsquigarrow h' \qquad \Delta \vdash \tau' = \tau : \mathrm{Type}}{\Psi; \Delta; \Gamma \vdash h \,\mathrm{at}\, r : \tau \rightsquigarrow h'}$$

$\mathcal{H}(\mathrm{fix}\, f[\Delta](C, x_1 : \tau_1, \ldots x_n : \tau_n).e \,\mathrm{at}\, r) =$
$\quad \mathrm{fix}\, f[\mathbf{\Delta}(\Delta)]([\Delta]\mathcal{C}(C), x_1 : [\Delta]\mathcal{T}(\tau_1), \ldots x_n : [\Delta]\mathcal{T}(\tau_n)).\mathcal{E}(e)$
$\mathcal{H}(\langle v_1, \ldots, v_n \rangle \,\mathrm{at}\, r) = \langle \mathcal{V}(v_1), \ldots, \mathcal{V}(v_n) \rangle$
$\mathcal{D}(x = v) = (x = \mathcal{V}(v))$
$\mathcal{D}(x = v_1 \, p \, v_2) = (x = \mathcal{V}(v_1) \, p \, \mathcal{V}(v_2))$

$$\frac{\Psi; \Delta; \Gamma \vdash v : r \,\mathrm{handle} \rightsquigarrow v' \qquad \Psi; \Delta; \Gamma \vdash h \,\mathrm{at}\, r : \tau \rightsquigarrow h' \qquad \Delta \vdash C \leq C' \oplus \{r^+\}}{\Psi; \Delta; \Gamma; C \vdash x = h \,\mathrm{at}\, v \Longrightarrow \Delta; \Gamma, x : \tau; C \rightsquigarrow (x = h' \,\mathrm{at}\, v')} (x \notin \mathit{domain}(\Gamma))$$

$\mathcal{D}(x = \#n \, v) = (x = \#n \, \mathcal{V}(v))$
$\mathcal{D}(\mathrm{newrgn}\, \rho, x) = \mathrm{newrgn}\, \rho, x$
$\mathcal{D}(\mathrm{freergn}\, v) = \mathrm{freergn}\, \mathcal{V}(v)$

$\mathcal{E}(\text{let}\, d\, \text{in}\, e) = \text{let}\, \mathcal{D}(d)\, \text{in}\, \mathcal{E}(e)$
$\mathcal{E}(\text{if0}\, v\, \text{then}\, e_2\, \text{else}\, e_3) = \text{if0}\, \mathcal{V}(v)\, \text{then}\, \mathcal{E}(e_2)\, \text{else}\, \mathcal{E}(e_3)$
$\mathcal{E}(v(v_1, \ldots, v_n) = \mathcal{V}(v)(\mathcal{V}(v_1), \ldots, \mathcal{V}(v_n)))$
$\mathcal{E}(\text{halt}\, v) = \text{halt}\, \mathcal{V}(v)$

$$\frac{\vdash \Psi \qquad \Psi \vdash R_1\, \text{at}\, \nu_1 : \Upsilon_1 \rightsquigarrow R_1' \qquad \ldots \qquad \Psi \vdash R_n\, \text{at}\, \nu_n : \Upsilon_n \rightsquigarrow R_n' \qquad \Psi = \{\nu_1 : \Upsilon_1, \ldots, \nu_n : \Upsilon_n\}}{\vdash \{\nu_1 \mapsto R_1, \ldots, \nu_n \mapsto R_n\} : \Psi \rightsquigarrow \{\nu_1 \mapsto R_1', \ldots, \nu_n \mapsto R_n'\}}$$

$$\frac{\Psi; \cdot; \cdot \vdash h_1\, \text{at}\, \nu : \tau_1 \rightsquigarrow h_1' \qquad \ldots \qquad \Psi; \cdot; \cdot \vdash h_n\, \text{at}\, \nu : \tau_n \rightsquigarrow h_n'}{\Psi \vdash \{\ell_1 \mapsto h_1, \ldots, \ell_n \mapsto h_n\}\, \text{at}\, \nu : \{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\} \rightsquigarrow \{\ell_1 \mapsto h_1', \ldots, \ell_n \mapsto h_n'\}}$$

D.4.1 Lemmas for CC/CCL→CC/SLL

- If $\vdash \Delta$ and $\Delta \vdash r : \text{Res}$ then $[\Delta]r = r$
- If $\vdash \Delta$ and $\Delta \vdash C : \text{Cap}$ then $\mathbf{\Delta}(\Delta) \vdash [\Delta]\mathcal{C}(C) : \text{Cap}$
- If $\vdash \Delta$ and $\Delta \vdash C_1 = C_2 : \text{Cap}$ then $\mathbf{\Delta}(\Delta) \vdash [\Delta]\mathcal{C}(C_1) = [\Delta]\mathcal{C}(C_2) : \text{Cap}$
- If $\vdash \Delta$ and $\Delta \vdash C_1 \leq C_2$ then $\mathbf{\Delta}(\Delta) \vdash [\Delta]\mathcal{C}(C_1) \leq [\Delta]\mathcal{C}(C_2)$
  - If in addition $\mathbf{\Delta}^1(\Delta) \vdash \mathcal{C}^1(C_1) \leq \mathcal{C}^1(C_2) \rightsquigarrow U$ then $\mathbf{\Delta}(\Delta) \vdash [\Delta]\mathcal{U}^3(\mathcal{U}^2(\mathcal{C}^1(C_1))) = [\Delta]\mathcal{U}^3(U \oplus \mathcal{U}^2(\mathcal{C}^1(C_2))) : \text{Cap}$ and $\mathbf{\Delta}(\Delta) \vdash [\Delta]\mathcal{U}^3(U) \otimes [\Delta]\mathcal{A}^3(\mathcal{A}^2(\mathcal{C}^1(C_1))) \otimes \text{true} \leq [\Delta]\mathcal{A}^3(\mathcal{A}^2(\mathcal{C}^1(C_2))) \otimes \text{true}$
- If $\vdash \Delta$ and $\Delta \vdash C = C' \oplus \{r^1\} : \text{Cap}$ then $\mathbf{\Delta}(\Delta) \vdash [\Delta]\mathcal{C}(C) = ([\Delta]\mathcal{C}(C')) \otimes \{r\} : \text{Cap}$
- If $\vdash \Delta$ and $\Delta \vdash C \leq C' \oplus \{r^+\}$ then $\mathbf{\Delta}(\Delta) \vdash [\Delta]\mathcal{C}(C) \leq \{r\} \otimes \text{true}$
- If $\vdash \Delta$ then $\vdash [\Delta]\mathcal{C}(\emptyset)$
- If $\vdash \Delta$ and $\Delta \vdash \tau : \text{Type}$ then $\mathbf{\Delta}(\Delta) \vdash [\Delta]\mathcal{T}(\tau) : \text{Type}$
- If $\vdash \Delta$ and $\Delta \vdash \tau_1 = \tau_2 : \text{Type}$ then $\mathbf{\Delta}(\Delta) \vdash [\Delta]\mathcal{T}(\tau_1) = [\Delta]\mathcal{T}(\tau_2) : \text{Type}$
- If $\vdash \Psi$ and $\Psi(\nu.\ell)$ exists then $\mathbf{\Psi}(\Psi)(\nu.\ell) = [\Delta]\mathcal{T}(\Psi(\nu.\ell))$
- If $\vdash \Delta$ and $\Delta \vdash \alpha' : \text{Cap}$ and $\Delta \vdash C : \text{Cap}$ then $\mathcal{C}([\alpha' \leftarrow C']C) = [\alpha'_U \leftarrow \mathcal{U}^3(\mathcal{U}^2(\mathcal{C}^1(C'))), \alpha_A \leftarrow \mathcal{A}^3(\mathcal{A}^2(\mathcal{C}^1(C'))), \alpha'_{SU} \leftarrow \mathcal{U}^3(\mathcal{U}^2(\mathcal{S}^1(C'))), \alpha_{SA} \leftarrow \mathcal{A}^3(\mathcal{A}^2(\mathcal{S}^1(C')))]\mathcal{C}(C)$
- If $\vdash \Delta$ and $\Delta \vdash \alpha' : \text{Cap}$ and $\Delta \vdash \tau : \text{Type}$ then $\mathcal{T}([\alpha' \leftarrow C']\tau) = [\alpha'_U \leftarrow \mathcal{U}^3(\mathcal{U}^2(\mathcal{C}^1(C'))), \alpha_A \leftarrow \mathcal{A}^3(\mathcal{A}^2(\mathcal{C}^1(C'))), \alpha'_{SU} \leftarrow \mathcal{U}^3(\mathcal{U}^2(\mathcal{S}^1(C'))), \alpha_{SA} \leftarrow \mathcal{A}^3(\mathcal{A}^2(\mathcal{S}^1(C')))]\mathcal{T}(\tau)$
- If $\vdash \Delta$ and $\Delta \vdash \alpha' : \text{Type}$ and $\Delta \vdash \tau : \text{Type}$ then $\mathcal{T}([\alpha' \leftarrow \tau']\tau) = [\alpha' \leftarrow \mathcal{T}(\tau')]\mathcal{T}(\tau)$
- If $\vdash \Psi$ and $\vdash \Delta$ and $\Delta \vdash \Gamma$ then:
  - If $\Psi; \Delta; \Gamma \vdash v : \tau \rightsquigarrow v'$ then $\mathbf{\Psi}(\Psi); \mathbf{\Delta}(\Delta); [\Delta]\mathbf{\Gamma}(\Gamma) \vdash [\Delta]v' : [\Delta]\mathcal{T}(\tau)$
  - If $\Psi; \Delta; \Gamma \vdash h\, \text{at}\, r : \tau \rightsquigarrow h'$ then $\mathbf{\Psi}(\Psi); \mathbf{\Delta}(\Delta); [\Delta]\mathbf{\Gamma}(\Gamma) \vdash [\Delta]h'\, \text{at}\, r : [\Delta]\mathcal{T}(\tau)$
  - If $\Psi; \Delta; \Gamma; C \vdash d \implies \Delta'; \Gamma'; C' \rightsquigarrow d'$ then $\mathbf{\Psi}(\Psi); \mathbf{\Delta}(\Delta); [\Delta]\mathbf{\Gamma}(\Gamma); [\Delta]\mathcal{C}(C) \vdash [\Delta]d' \implies \mathbf{\Delta}(\Delta'); [\Delta]\mathbf{\Gamma}(\Gamma'); [\Delta]\mathcal{C}(C')$ and $[\Delta] = [\Delta']$
  - If $\Psi; \Delta; \Gamma; C \vdash e \rightsquigarrow e'$ then $\mathbf{\Psi}(\Psi); \mathbf{\Delta}(\Delta); [\Delta]\mathbf{\Gamma}(\Gamma); [\Delta]\mathcal{C}(C) \vdash [\Delta]e'$
- If $\vdash \Psi$ and $\Delta \vdash M : \Psi \rightsquigarrow M'$ and $\Psi \vdash C\, \mathit{sat}$ and $\Psi; \{\}; \{\}; C \vdash e \rightsquigarrow e'$ then $\vdash M' : \mathbf{\Psi}(\Psi)$ and $\mathbf{\Psi}(\Psi) \vdash \mathcal{C}(C)\, \mathit{sat}$ and $\mathbf{\Psi}(\Psi); \{\}; \{\}; \mathcal{C}(C) \vdash e'$

For proofs, see [12].