Lightweight Linear Types in System F°

Karl Mazurak Jianzhou Zhao Steve Zdancewic
University of Pennsylvania
{mazurak,jianzhou,stevez}@cis.upenn.edu

Abstract

We present System F°, an extension of System F that uses *kinds* to distinguish between linear and unrestricted types, simplifying the use of linearity for general-purpose programming. We demonstrate through examples how System F° can elegantly express many useful protocols, and we prove that any protocol representable as a DFA can be encoded as an F° type. We supply mechanized proofs of System F°'s soundness and parametricity properties, along with a nonstandard operational semantics that formalizes common intuitions about linearity and aids in reasoning about protocols.

We compare System F° to other linear systems, noting that the simplicity of our kind-based approach leads to a more explicit account of what linearity is meant to capture, allowing otherwise-conflicting interpretations of linearity (in particular, restrictions on *aliasing* versus restrictions on resource *usage*) to coexist peacefully. We also discuss extensions to System F° aimed at making the core language more practical, including the additive fragment of linear logic, algebraic datatypes, and recursion.

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

General Terms Design, Languages, Theory

Keywords Linear logic, Polymorphism, Type systems

1. Introduction

Linear logic [15, 16] models resource usage by restricting the properties of contraction (the ability to duplicate a resource) and weakening (the ability to discard a resource). In the context of programming languages, ideas from linear logic were quickly adopted, at first to eliminate garbage collection [21] and shortly thereafter to handle mutable state [29].

Since their introduction, variants, refinements, and improvements on linear type systems have been proposed for many applications, including explicit memory management and control of aliasing [2, 13, 17, 28, 36], capabilities [9, 10], and tracking state changes for program analysis [11, 32]. Of particular interest is work on *typestates*, which ensure that a sequence of API calls is well-behaved [13, 12], and on *session types*, which check that the endpoints of a channel agree on the next message to be sent or received [18, 25, 27]. Walker has a more comprehensive survey [33].

TLDI'10 January 23, 2010, Madrid, Spain.

Given these success stories, it is perhaps surprising that we have yet to see general linear types seriously considered for inclusion in a mainstream functional programming language.[1] But alas, linear types can easily lead to awkward programming models and potentially complicated language designs that are difficult both to implement and to program with. This paper seeks to address these issues by introducing System F°—pronounced "F-pop"—a language that is intended to be a simple foundation for practical linear programming. Rather than aiming at one particular problem, System F° lets programmers enforce their own protocol abstractions through the power of linearity and polymorphism, yet its typing discipline is lightweight enough to expose in a surface language.

System F° is simply the Girard–Reynolds polymorphic λ-calculus [14, 23] extended with two base kinds: $\star$, classifying ordinary, unrestricted types, and $\circ$, classifying linear types. A subkinding relation $\star \leq \circ$ makes explicit the observation that values of unrestricted types may safely be treated linearly; *i.e.*, since variables of unrestricted type may be used any number of times and linear variables must be used exactly once, it is always safe to use unrestricted variables as though they were linear. Any System F° expression with kinds erased is a well-typed System F expression.

In introducing System F°, this paper contributes the following:

- The design of System F°, and in particular its use of kinds and kind subsumption, which is lightweight and structured so as to integrate well with existing functional languages. (Section 2)
- Mechanized proofs of standard soundness and parametricity results for System F°, which ensure that the properties functional programmers rely on continue to hold. (Section 2.1)
- A second, linearity-aware semantics for System F°, which formalizes common intuitions about linearity and shows that these intuitions do indeed hold. (Section 4)
- Several examples—including all regular languages—along with extensions and proposals for compiler support, remarkable primarily for their simplicity, which showcase System F°'s potential usefulness. (Sections 3 and 5)

In the rest of this section we discuss the design of System F° in the context of prior work, and we showcase System F°'s ability to enforce programmer-defined protocols with a familiar example.

1.1 Prior work: Linear type system design considerations

There are many variants on linear type systems in the literature [33], but the crucial design decision from our perspective is how linear and unrestricted variables are differentiated and how that mechanism interacts with polymorphism. Our use of kinds and kind subsumption is intended to capture the essence of linearity simply and generally while remaining faithful to the standard

[1] Clean is often seen as the exception to this, but there are subtle differences between its uniqueness types, which concern aliasing, and standard linear types (see Section 3.1).

semantics and programming model of System F. This division between linear and unrestricted types is inspired by that proposed for *monomorphic* systems by Wadler [29] and Benton [5]. Here we contrast our approach with some alternatives from the literature.

Broadly speaking, there are two other approaches to distinguishing linear from unrestricted variables. The first approach, which closely follows linear logic, is to treat *all* types as linear and introduce the modal constructor ! to account for unrestricted terms, which must be closed with respect to linear variables. This interacts easily enough with polymorphism, since all types are treated uniformly—any type can be substituted for any type variable.

Unfortunately, assuming linearity by default requires very explicit handling of unrestricted terms; the standard approach uses a **let** ! construct to introduce unrestricted variables from suspended computations $!e$ [6, 30]. Because unrestricted values are common in practice, this can prove quite cumbersome, and the burden extends even to base types—the constant 3 would be of type Int, not $!\mathsf{Int}$. Further, full inference of !s has been shown to be impossible [22].

A second approach distinguishes linear from unrestricted types by means of *qualifiers* lin (for "linear") and un (for "unrestricted") applied to a collection of pre-types [1, 33]. These qualifiers (which do not nest) constrain usage (or aliasing), while the pre-types determine the introduction and elimination forms of values. This separation facilitates implicit copying and discarding for unrestricted types, yielding a less burdensome programming model.

Type qualifiers, however, have more complex interactions with polymorphism. To retain both soundness and expressivity, one is led to introduce quantification over qualifiers, pre-types, and types independently. This quickly leads to large and complex types; for instance, the type of plus has five qualifiers:

$$\mathsf{plus} : (q_1\ \mathsf{Int}) \xrightarrow{q_4} (q_2\ \mathsf{Int}) \xrightarrow{q_5} (q_3\ \mathsf{Int})$$

The relationships among such qualifiers are often nontrivial (*e.g.* q_5 should be lin if q_1 is), which can be captured (at the expense of additional complexity) by qualifier-level bounded quantification.[2] Qualifiers thus ease the use of unrestricted types but are too unwieldy for a polymorphic source language—indeed, others have argued against qualifiers even for intermediate languages [34].

Our use of kinds in lieu of ! or type qualifiers strikes a good balance on these issues. As with qualifiers, programming with unrestricted types is natural; as with !, polymorphism remains simple. Subkinding also plays well with base types: 3 has type Int, which has kind $\star$ (and, by subsumption, $\circ$), while the type of plus is the simple $\mathsf{Int} \xrightarrow{\star} \mathsf{Int} \xrightarrow{\star} \mathsf{Int}$. We thus have flexible polymorphic types without the need for bounded quantification or other complexities of subtyping in a higher-order setting.

Closest in design to System F$^\circ$ is probably Ahmed, Fluet, and Morrisett's language for substructural state [1], though they use qualifiers as described above. Due to their focus on aliasing in mutable state, their language does not admit subtyping, which would be the analog in their setting of our subkinds. Our subkind relation $\star \leq \circ$ agrees with the interpretation of linearity as related to *usage*; it does not reflect linearity as *alias-freedom*, in which linear types are analogous to uniqueness types [17, 28]. Nevertheless, we show in Section 3.1 that matters of aliasing can indeed be addressed in System F$^\circ$ given an appropriate representation of references.

Ahmed et al. also admit *affine types*, for which only weakening is permitted, and *relevant types*, which allow only contraction. Such concepts would fit well with our subkinding relation, but would increase the complexity of typing context management. Other systems have considered notions of usage much more fine-grained than ours [19], but the types in such systems can quickly become overwhelming if exposed to the programmer. As we are incorporating the full power of System F, exposing at least some types is unavoidable; we also believe that System F$^\circ$ convincingly demonstrates that just the simple distinction between linear and unrestricted types has much to offer.

[2] Making Int a proper type rather than a pre-type simplifies the type of plus but prohibits certain polymorphic functions from accepting integers.

1.2 System F$^\circ$ by example: Types for filehandles

Linearity lets us specify a filesystem interface that requires a filehandle to be closed exactly once and forbids its use thereafter. A first approximation[3] for an idealized linear filesystem might be

$$\begin{aligned}
\mathsf{FileHandle} &: \circ \\
\\
\mathsf{open} &: \mathsf{String} \xrightarrow{\star} \mathsf{FileHandle} \\
\mathsf{read} &: \mathsf{FileHandle} \xrightarrow{\star} (\mathsf{Char}, \mathsf{FileHandle}) \\
\mathsf{write} &: \mathsf{Char} \xrightarrow{\star} \mathsf{FileHandle} \xrightarrow{\star} \mathsf{FileHandle} \\
\mathsf{close} &: \mathsf{FileHandle} \xrightarrow{\star} \mathsf{Unit} \\
\\
\mathsf{readLine} &: \mathsf{FileHandle} \xrightarrow{\star} (\mathsf{String}, \mathsf{FileHandle})
\end{aligned}$$

Here open, read, write, and close are intended to be primitive operations over the linear (note the kind ascription) type $\mathsf{FileHandle}$, while $\mathsf{readLine}$ is one of many library functions defined to make file access more convenient. The $\star$ decorating the arrow in a type like $\mathsf{open} : \mathsf{String} \xrightarrow{\star} \mathsf{FileHandle}$ indicates that open is an unrestricted function, which may be used more than once; each time open is invoked, it will return a new $\mathsf{FileHandle}$ value that must be used linearly. Unrestricted functions are also free to take arguments of linear type, as can be seen in the other operations. Similarly, a linear function of type $\tau_1 \xrightarrow{\circ} \tau_2$ should be invoked exactly once, but this has no bearing on the kind of either τ_1 or τ_2.

Because operations like read and write consume a $\mathsf{FileHandle}$ as input and return a linear $\mathsf{FileHandle}$ as output, and because such values cannot be duplicated, client programs are forced to sequence the calls to these functions, fixing their order of evaluation. Since $\mathsf{FileHandle}$ values cannot be discarded, the program—unless its overall type indicates that it contains a filehandle—must eventually use close to dispose of any $\mathsf{FileHandle}$s it has created. Linearity ensures that no aliased or duplicated $\mathsf{FileHandle}$s representing closed files remain to be improperly accessed. Clients of this interface are thus constrained, after opening a file, to access that file according to the regular protocol $(\mathsf{read}|\mathsf{write})^*\mathsf{close}$.

Unfortunately, today's operating systems do not understand linearity and instead provide unrestricted interfaces, which make no guarantees about correct filehandle usage:

$$\begin{aligned}
\mathsf{UnsafeFH} &: \star \\
\\
\mathsf{unsafeOpen} &: \mathsf{String} \xrightarrow{\star} \mathsf{UnsafeFH} \\
\mathsf{unsafeRead} &: \mathsf{UnsafeFH} \xrightarrow{\star} \mathsf{Char} \\
\mathsf{unsafeWrite} &: \mathsf{Char} \xrightarrow{\star} \mathsf{UnsafeFH} \xrightarrow{\star} \mathsf{Unit} \\
\mathsf{unsafeClose} &: \mathsf{UnsafeFH} \xrightarrow{\star} \mathsf{Unit}
\end{aligned}$$

System F$^\circ$ makes it easy to create a safe interface protecting the above from misuse. First, for α of kind $\circ$—our abstract representation of an actual filehandle—we define a record of safe file operations:

$$\begin{aligned}
\mathsf{File}(\alpha) \;=\; \{\; & \mathsf{read} : \alpha \xrightarrow{\star} (\mathsf{Char}, \alpha), \\
& \mathsf{write} : \mathsf{Char} \xrightarrow{\star} \alpha \xrightarrow{\star} \alpha, \\
& \mathsf{close} : \alpha \xrightarrow{\star} \mathsf{Unit} \;\}
\end{aligned}$$

[3] This example uses roughly the same interface given by DeLine and Fähndrich to motivate Vault [11] and discussed by Kiselyov and Shan [20] as an alternative to their filehandle regions.

$$
\begin{array}{rcll}
\kappa & ::= & \star \mid \circ & \textit{kinds} \\
\tau & ::= & \alpha \mid \tau \xrightarrow{\kappa} \tau \mid \forall \alpha{:}\kappa.\ \tau & \textit{types} \\
\\
e & ::= & x \mid \lambda^{\kappa} x{:}\tau.\ e \mid e\ e \mid \Lambda \alpha{:}\kappa.\ v \mid e\ [\tau] & \textit{expressions} \\
v & ::= & \lambda^{\kappa} x{:}\tau.\ e \mid \Lambda \alpha{:}\kappa.\ v & \textit{values} \\
\\
\Gamma & ::= & \cdot \mid \Gamma, \alpha{:}\kappa \mid \Gamma, x{:}\tau & \textit{unrestricted typing contexts} \\
\Delta & ::= & \cdot \mid \Delta, x{:}\tau & \textit{linear typing contexts}
\end{array}
$$

Figure 1. System F$^\circ$

Now we can define open to return both the hidden filehandle and its associated operations:

$$
\begin{array}{l}
\mathsf{open}\ :\ \mathsf{String} \xrightarrow{\star} \exists \alpha{:}\circ.\ (\alpha, \mathsf{File}(\alpha)) \\
\mathsf{open}\ =\ \lambda^{\star} f{:}\mathsf{String}.\ \textbf{let}\ \textit{handle} = \mathsf{unsafeOpen}\, f\ \textbf{in} \\
\qquad \textbf{pack}\ \alpha = \mathsf{UnsafeFH}\ \textbf{in} \\
\qquad (\ \textit{handle}, \\
\qquad\quad \{\ \mathsf{read} = \lambda^{\star} h{:}\mathsf{UnsafeFH}.\ (\mathsf{unsafeRead}\ h, h), \\
\qquad\quad\ \ \mathsf{write} = \lambda^{\star} c{:}\mathsf{Char}.\ \lambda^{\star} h{:}\mathsf{UnsafeFH}. \\
\qquad\quad\qquad\qquad \mathsf{unsafeWrite}\ c\ h; h, \\
\qquad\quad\ \ \mathsf{close} = \mathsf{unsafeClose} \\
\qquad\quad \}\ ) : (\alpha, \mathsf{File}(\alpha))
\end{array}
$$

Note that, while the type UnsafeFH is unrestricted within the scope of open, the outside world sees its occurrences at the existentially bound linear type variable α. If open treats filehandles correctly, then any use of an existential package created by open must treat them correctly as well.

We no longer read, write, and close as separate functions, but library functions like readLine, are still useful to have. We could simply replace the FileHandles in the first type proposed for readLine with open's return type, $\exists \alpha{:}\circ.\ (\mathsf{File}(\alpha), \alpha)$, but a smarter choice is

$$\mathsf{readLine} : \forall \alpha{:}\circ.\ \mathsf{File}(\alpha) \xrightarrow{\star} \alpha \xrightarrow{\star} (\mathsf{String}, \alpha)$$

This makes clear that $\mathsf{File}(\alpha)$ contains only the (unrestricted) file operations, not the filehandle itself, and by separating out **unpack**s and **pack**s from calls to both primitive operations and library functions, our types can reflect the fact that the filehandle returned by readLine is the same one that it was given. Writing our functions this way allows for useful type coercions; for example, suppose we are also able to open files in read-only mode, resulting in restricted existential packages of the form

$$
\begin{array}{rcl}
\mathsf{ROFile}(\alpha) & = & \{\ \mathsf{read} : \alpha \xrightarrow{\star} (\mathsf{Char}, \alpha), \\
& & \qquad \mathsf{close} : \alpha \xrightarrow{\star} \mathsf{Unit}\ \}
\end{array}
$$

Many functions, including readLine, may be defined over this weaker interface, and we can always construct a record of type $\mathsf{ROFile}(\alpha)$ out of a record of type $\mathsf{File}(\alpha)$ to allow a read-write filehandle to be treated as though it were read-only.

Of course, such a filesystem interface is of little use if it is too cumbersome for everyday programming. In Section 5.2 we discuss modest compiler support that ensures that this is not the case.

2. System F$^\circ$ Defined

The syntax of System F$^\circ$ is given in Figure 1; Figure 2 shows its call-by-value (and kind-agnostic) operational semantics, which is completely standard. Type variables are annotated with their kinds when bound (by Λ in expressions or by $\forall$ in types); kinds also appear in functions ($\lambda^{\kappa} x{:}\tau.\ e$) and function types ($\tau_1 \xrightarrow{\kappa} \tau_2$). F$^\circ$ adopts the *value restriction* [35], permitting type abstraction only over values, for reasons that will become clear later.

Typing and kinding rules for System F$^\circ$, along with auxiliary judgments, are given in Figure 3. Our typing rules take a linear

$$[\text{E-AppLam}]\ (\lambda^{\kappa} x{:}\tau.\ e)\ v \longrightarrow \{x \mapsto v\}e$$

$$[\text{E-TAppLam}]\ (\Lambda \alpha{:}\kappa.\ v)\ [\tau] \longrightarrow \{\alpha \mapsto \tau\}v$$

$$[\text{E-App1}]\ \frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2} \qquad [\text{E-App2}]\ \frac{e \longrightarrow e'}{v\ e \longrightarrow v\ e'}$$

$$[\text{E-TApp}]\ \frac{e \longrightarrow e'}{e\ [\tau] \longrightarrow e'\ [\tau]}$$

Figure 2. Evaluation rules for System F$^\circ$

typing context Δ, binding only term variables, in addition to the standard unrestricted context Γ, which binds both type and term variables; following Barber's DILL [3], this greatly simplifies the mechanization of our soundness proofs. At the kind level, note that the rule K-Arrow gives $\tau_1 \xrightarrow{\kappa} \tau_2$ the kind κ regardless of the kinds of τ_1 and τ_2. As described in Section 1.2, this decouples the notion of a linear function (i.e., of type $\tau_1 \xrightarrow{\circ} \tau_2$), which must be used exactly once, from a function that takes a linear argument (i.e., where τ_1 has kind $\circ$), which must use its argument exactly once. Rule K-All, by contrast, simply gives $\forall \alpha{:}\kappa.\tau$ the same kind as τ—this is a design choice made in the interests of keeping F$^\circ$ simple, as little seems to be gained by allowing these kinds to differ; this choice is also compatible with a type-erasure interpretation.

Type application must check that the kind of the supplied type argument is compatible with the kind of the variable for which it will be substituted. As mentioned in Section 1, we see linearity as constraining the permitted usage of a variable, which means that it should always be safe to replace a linear type variable with an unrestricted type; we thus allow subkinding via the rule K-Sub. Subsumption turns out to be key in many useful examples: for instance, it is what allowed the unrestricted type UnsafeFH to be protected by a linear interface.

Weakening—neglecting to use a variable—and contraction—using a variable more than once—should only be possible with variables of unrestricted type. Rather than add explicit contraction and weakening rules, we have built these properties into the rules that require them. The separation of linear and unrestricted typing contexts makes this fairly straightforward; rules T-LVar and T-UVar permit weakening by allowing an arbitrary Γ at the leaves of typing derivations, while rule T-App duplicates Γ but splits Δ via the $\Cup$ relation. We can thus prove:[4]

Lemma 1 (Weakening)**.** *If* $\Gamma_1, \Gamma_2; \Delta \vdash e : \tau'$ *and* $\Gamma_1 \vdash \tau : \star$*, then* $\Gamma_1, x{:}\tau, \Gamma_2; \Delta \vdash e : \tau'$.

Lemma 2 (Contraction)**.** *If* $\Gamma_1, x{:}\tau, y{:}\tau, \Gamma_2; \Delta \vdash e : \tau'$ *and* $\Gamma_1 \vdash \tau : \star$*, then* $\Gamma_1, x{:}\tau, \Gamma_2; \Delta \vdash \{y \mapsto x\}e : \tau'$.

Linear variables must not inadvertently be captured by unrestricted function closures. To this end, rule T-Lam constrains its linear context Δ according to the λ's kind annotation κ: Δ must be empty if κ is $\star$. We thus ensure that an unrestricted function cannot capture linear variables in its closure, even though it may well take an argument of linear type.

Only unrestricted function arguments should be placed in Γ, but subkinding allows any type to be considered as linear and hence any expression variable to be bound in Δ—a fact which proves crucial in the proof of preservation. We write this potentially nondeterministic context extension as $[\Gamma; \Delta], x{:}\tau \ni \Gamma'; \Delta'$.

[4] The lemmas in this section of the paper have all been proved in Coq; the source scripts of our proofs are available from the last author's web pages.

$$[\text{K-Sub}]\ \frac{\Gamma \vdash \tau : \star}{\Gamma \vdash \tau : \circ} \qquad [\text{K-Arr}]\ \frac{\Gamma \vdash \tau_1 : \kappa_1 \qquad \Gamma \vdash \tau_2 : \kappa_2}{\Gamma \vdash \tau_1 \xrightarrow{\kappa} \tau_2 : \kappa} \qquad [\text{K-TVar}]\ \frac{\alpha{:}\kappa \in \Gamma}{\Gamma \vdash \alpha : \kappa} \qquad [\text{K-All}]\ \frac{\Gamma, \alpha{:}\kappa \vdash \tau : \kappa' \qquad \alpha \notin \Gamma}{\Gamma \vdash \forall \alpha{:}\kappa.\, \tau : \kappa'}$$

$$[\text{U-Empty}]\ \cdot \Cup \cdot = \cdot \qquad [\text{U-Left}]\ \frac{\Delta_1 \Cup \Delta_2 = \Delta \qquad x \notin \Delta}{\Delta_1, x{:}\tau \Cup \Delta_2 = \Delta, x{:}\tau} \qquad [\text{U-Right}]\ \frac{\Delta_1 \Cup \Delta_2 = \Delta \qquad x \notin \Delta}{\Delta_1 \Cup \Delta_2, x{:}\tau = \Delta, x{:}\tau}$$

$$[\text{B-Lin}]\ \frac{\Gamma \vdash \tau : \circ \qquad x \notin \Gamma, \Delta}{[\Gamma; \Delta], x{:}\tau \ni \Gamma; (\Delta, x{:}\tau)} \qquad [\text{B-Un}]\ \frac{\Gamma \vdash \tau : \star \qquad x \notin \Gamma, \Delta}{[\Gamma; \Delta], x{:}\tau \ni (\Gamma, x{:}\tau); \Delta} \qquad [\text{T-LVar}]\ \Gamma; x{:}\tau \vdash x : \tau \qquad [\text{T-UVar}]\ \frac{x{:}\tau \in \Gamma}{\Gamma; \cdot \vdash x : \tau}$$

$$[\text{T-Lam}]\ \frac{[\Gamma; \Delta], x{:}\tau_1 \ni \Gamma'; \Delta' \qquad \Gamma'; \Delta' \vdash e : \tau_2 \qquad \Delta = \cdot \vee \kappa = \circ}{\Gamma; \Delta \vdash \lambda^{\kappa} x{:}\tau_1.\, e : \tau_1 \xrightarrow{\kappa} \tau_2} \qquad [\text{T-TLam}]\ \frac{\Gamma, \alpha{:}\kappa; \Delta \vdash v : \tau \qquad \alpha \notin \Gamma}{\Gamma; \Delta \vdash \Lambda \alpha{:}\kappa.\, v : \forall \alpha{:}\kappa.\, \tau}$$

$$[\text{T-App}]\ \frac{\Gamma; \Delta_1 \vdash e_1 : \tau_1 \xrightarrow{\kappa} \tau_2 \qquad \Gamma; \Delta_2 \vdash e_2 : \tau_1 \qquad \Delta_1 \Cup \Delta_2 = \Delta}{\Gamma; \Delta \vdash e_1\ e_2 : \tau_2} \qquad [\text{T-TApp}]\ \frac{\Gamma; \Delta \vdash e : \forall \alpha{:}\kappa.\, \tau' \qquad \Gamma \vdash \tau : \kappa}{\Gamma; \Delta \vdash e\ [\tau] : \{\alpha \mapsto \tau\}\tau'}$$

Figure 3. Kinding and typing rules for System F$^\circ$

let $x = e$ **in** e'	$\triangleq$	$(\lambda^\circ x{:}\tau.\, e')\ e$ *where e has type τ*
Unit	$\triangleq$	$\forall \alpha{:}\circ.\, \alpha \xrightarrow{\star} \alpha$
unit	$\triangleq$	$\Lambda \alpha{:}\circ.\, \lambda^\star x{:}\alpha.\, x$
$e_1; e_2$	$\triangleq$	**let** $\_ = e_1$ **in** e_2
(τ_1, τ_2)	$\triangleq$	$\forall \alpha{:}\circ.\, (\tau_1 \xrightarrow{\circ} \tau_2 \xrightarrow{\circ} \alpha) \xrightarrow{\circ} \alpha$
$(,)$	$\triangleq$	$\Lambda \alpha{:}\circ.\, \Lambda \beta{:}\circ.\, \lambda^\star x{:}\alpha.\, \lambda^\circ y{:}\beta.$ $\Lambda \gamma{:}\circ.\, \lambda^\circ f{:}\alpha \xrightarrow{\circ} \beta \xrightarrow{\circ} \gamma.\, f\ x\ y$
let $(x, y) = e$ **in** e'	$\triangleq$	$e\ [\tau']\ (\lambda^\circ x{:}\tau_1.\, \lambda^\circ y{:}\tau_2.\, e')$ *where e' has type τ'*
$\exists \alpha{:}\kappa.\, \tau'$	$\triangleq$	$\forall \beta{:}\circ.\, (\forall \alpha{:}\kappa.\, \tau' \xrightarrow{\circ} \beta) \xrightarrow{\kappa'} \beta$
pack $\alpha{:}\kappa = \tau$ **in** $e : \tau'$	$\triangleq$	**let** $x = e$ **in** $\Lambda \beta{:}\circ.\, \lambda^{\kappa'} f{:}(\forall \alpha{:}\kappa.\, \tau' \xrightarrow{\circ} \beta).\, f\ [\tau]\ x$
unpack $\alpha, x = e$ **in** e'	$\triangleq$	$e\ [\tau']\ (\Lambda \alpha{:}\kappa.\, \lambda^\circ x{:}\tau.\, e')$ *where e' has type τ'*

Figure 4. System F encodings in F$^\circ$

It is easy to see that, modulo the value restriction, System F$^\circ$ is an extension of System F. With this in mind, Figure 4 gives several well-known System F encodings that we make use of[5]. Aside from kind annotations, these are all standard; the linear annotations on type variables are for generality—the pairs so encoded are linear, for instance—while those that are on arrows account for captured linear variables in the arguments.

2.1 Metatheory of System F$^\circ$

Type Soundness We have verified in Coq that System F$^\circ$ enjoys type safety—a crucial but unsurprising result, given its similarity to System F. As is standard, we define soundness in terms of two properties: progress and preservation. Progress, which states that a closed, well-typed non-value can always take an evaluation step, is no different than in ordinary System F:

[5] We also make use of records of unrestricted type; their encodings would generalize that of pairs, but with more $\star$ annotations.

Lemma 3 (Progress)**.** *If $\cdot; \cdot \vdash e : \tau$, then either e is a value or there exists some e' such that $e \longrightarrow e'$.*

Proof. Induction on typing derivations, completely standard. □

Preservation, on the other hand, requires a bit of care. As usual, it depends on various substitution lemmas, and we must keep linearity in mind when formulating them.

Lemma 4 (Substitution)**.**

1. *If $\Gamma_1, \alpha{:}\kappa', \Gamma_2 \vdash \tau : \kappa$ and $\Gamma_1 \vdash \tau' : \kappa'$ then $\Gamma_1, \{\alpha \mapsto \tau'\}\Gamma_2 \vdash \{\alpha \mapsto \tau'\}\tau : \kappa$.*
2. *If $\Gamma_1, \alpha{:}\kappa', \Gamma_2; \Delta \vdash e : \tau$ and $\Gamma_1 \vdash \tau' : \kappa'$ then $\Gamma_1, \{\alpha \mapsto \tau'\}\Gamma_2; \{\alpha \mapsto \tau'\}\Delta \vdash \{\alpha \mapsto \tau'\}e : \{\alpha \mapsto \tau'\}\tau$.*
3. *If $\Gamma_1, x{:}\tau', \Gamma_2; \Delta \vdash e : \tau$ and $\Gamma_1; \cdot \vdash e' : \tau'$ then $\Gamma_1, \Gamma_2; \Delta \vdash \{x \mapsto e'\}e : \tau$.*
4. *If $\Gamma; \Delta_1, x : \tau', \Delta_2 \vdash e : \tau$ and $\Gamma; \Delta' \vdash e' : \tau'$ then $\Gamma; \Delta_1, \Delta', \Delta_2 \vdash \{x \mapsto e'\}e : \tau$.*

Proof. Each case by induction on the first derivation. The result relies heavily on various strengthening, weakening, and permutation lemmas (with respect to typing, context well-formedness, and the $\Cup$ relation) regarding the handling of typing contexts. □

Note that Substitution (3) does *not* hold if e' is permitted to contain free linear variables. Call-by-value reduction allows us to consider only values, however, and—because we have adopted the value restriction—we can prove that unrestricted values contain no free linear variables:

Lemma 5. *If $\Gamma; \Delta \vdash v : \tau$ and $\Gamma \vdash \tau : \star$ then $\Delta = \cdot$.*

Lemma 6 (Preservation)**.** *If $\Gamma; \Delta \vdash e : \tau$ and $e \longrightarrow e'$, then $\Gamma; \Delta \vdash e' : \tau$.*

From preservation and progress, soundness follows naturally:

Theorem 7 (Type soundness)**.** *If $\cdot; \cdot \vdash e{:}\tau$, then it is never the case that $e \longrightarrow^* e'$ where e' is not a value but cannot step further.*

Strong normalization and parametricity System F also has other properties of interest: it is *strongly normalizing*—evaluation always eventually reaches a value—and it enjoys relational *parametricity* [24]. We are able to cheat somewhat in proving the former for F$^\circ$ by observing that a well-typed System F$^\circ$ expression becomes a well-typed System F expression upon erasure of kind annotations.

As the two systems have identical operational behavior, strong normalization follows immediately.

Parametricity for F$^\circ$ cannot be proved by such an erasure, but, as one might hope from the similarity to System F, it is possible to directly adapt the standard logical relations proof with only minor syntactic differences due to our separation of unrestricted and linear contexts. We have thus proved (in Coq), for the standard relation between type substitutions $\rho_1 \approx \rho_2 : \Gamma$, relation between term substitutions $\rho \vdash \gamma_1 \approx \gamma_2 : \Gamma; \Delta$, and computation closure $\mathcal{C}[\![\tau]\!]_\rho$ of relations induced by a type τ, where ρ maps type variables to term relations and pairs of types and Γ and Δ bind the variables in the domain of the various substitutions:

Lemma 8 (Parametricity)**.** *If* $\Gamma; \Delta \vdash e : \tau$*,* $\rho_1 \approx \rho_2 : \Gamma$*, and* $\rho \vdash \gamma_1 \approx \gamma_2 : \Gamma; \Delta$*, then* $(\rho_1(\gamma_1(e)), (\rho_2(\gamma_2(e))) \in \mathcal{C}[\![\tau]\!]_\rho$*.*

Succinctly, this means that an expression e with type τ, under appropriate closing substitutions, is related to itself by (the computation closure of) the relation induced logically by τ.

Of course, this is only the simplest parametricity result we could provide; it does not take into account the extensions we propose in Section 5, nor does it take advantage of linearity in any way. The interactions between linearity and relational parametricity have been explored by Birkedal et al. [8] and Bierman et al. [7, 6] have explored program equivalences in the presence of !, both of which suggest avenues for future investigation in the context of System F$^\circ$. Our appeals to parametricity in Section 3.2, however, require nothing beyond what we have proved.

We view the ease with which we can adapt standard results from System F to System F$^\circ$ as a significant benefit of this design. A direct correspondence between these metatheoretic properties and intuitions about what linearity provides is not immediately obvious, however. In Section 4 we will show, by means of elaborated operational semantics, that the restrictions required for our soundness proofs are exactly those needed to satisfy our intuitions.

2.2 Comparison to traditional formulations

In contrast to our approach, linear type systems are more traditionally presented without kinds, assuming linearity by default and using the modality ! to allow unrestricted variables (see, for example [3, 6]). For the polymorphic λ-calculus, this gives us

$$\begin{array}{rcl} \sigma & ::= & \alpha \mid \sigma \multimap \sigma \mid \forall\alpha.\, \sigma \mid\, !\sigma \\ \\ t & ::= & a \mid x \mid \lambda a{:}\sigma.\, t \mid t\ t \mid \Lambda\alpha.\, t \mid t\ [\sigma] \\ & \mid & !t \mid \textbf{let}\ !x = t\ \textbf{in}\ t \\ \\ \Phi & ::= & \cdot \mid \Phi, \alpha \mid \Phi, x{:}\sigma \\ \Psi & ::= & \cdot \mid \Psi, a{:}\sigma \end{array}$$

For clarity, we distinguish between linear variables a, bound by λ terms, and non-linear variables x, bound by **let** !. The interesting typing rules concern the ! modality:

$$\frac{\Phi; \cdot \vdash t : \sigma}{\Phi; \cdot \vdash\, !t : !\sigma}$$

$$\frac{\Phi; \Psi_1 \vdash t_1 : !\sigma_1 \qquad \Phi, x{:}\sigma_1; \Psi_2 \vdash t_2 : \sigma_2 \qquad \Psi_1 \Cup \Psi_2 = \Psi}{\Phi; \Psi \vdash \textbf{let}\ !x = t_1\ \textbf{in}\ t_2 : \sigma_2}$$

In other words, the type $!\sigma$ indicates a term of type σ which uses no linear variables—the same constraint we place on unrestricted functions—and such a term can be captured by the **let** ! operation and subsequently used in an unrestricted fashion. Note, however, that the type $!\sigma$ itself is still linear, even though terms of that type allow for the introduction of unrestricted assumptions; while it is possible to formulate systems where terms of ! types can be duplicated or discarded directly, naive attempts to do so are unsound—for precisely the same reasons that, as discussed in Section 2.1 and Section 4, we require call-by-value reduction and the value restriction in System F$^\circ$—and sound formulations end up being heavier than those that make this distinction [31, 4].

We can encode the above system in ours easily enough; we first define a translation on types $[\![\sigma]\!]$ as

$$\begin{array}{rcl} [\![\alpha]\!] & = & \alpha \\ [\![\sigma_1 \multimap \sigma_2]\!] & = & [\![\sigma_1]\!] \xrightarrow{\circ} [\![\sigma_2]\!] \\ [\![\forall\alpha.\, \sigma]\!] & = & \forall\alpha{:}\circ.\, [\![\sigma]\!] \\ [\![!\sigma]\!] & = & \mathsf{Unit} \xrightarrow{\star} [\![\sigma]\!] \end{array}$$

The corresponding translation on terms is straightforward. The only interesting cases involve unrestricted variables and the ! modality:

$$\begin{array}{rcl} [\![x]\!] & = & x\ \mathsf{unit} \\ [\![!t]\!] & = & \lambda^\star\_{:}\mathsf{Unit}.\, [\![t]\!] \\ [\![\textbf{let}\ !x = t_1\ \textbf{in}\ t_2]\!] & = & (\lambda^\circ x{:}\mathsf{Unit} \xrightarrow{\star} [\![\sigma_1]\!].\, [\![t_2]\!])\ [\![t_1]\!] \\ & & \textit{where } t_1 \textit{ has type } !\sigma_1 \end{array}$$

Here we treat terms $!t$ as suspended computations which may be evaluated more than once—or not at all—which is standard.

Translating in the other direction is much less straightforward and space constraints preclude us from including the translation here. The translation on types is kind-directed and, at the term level, the insertion of ! and **let** ! operations is not trivial: a function that takes an unrestricted type must now take a ! type and bind it so that the variable need not appear linearly, but such arguments must be repackaged under ! in order to be passed to any subsequent functions. Polymorphic types and expressions also need care—for example, there are four cases needed to translate type instantiation, one for each combination of linear/unrestricted for the polymorphic term and its type argument.

We see this asymmetry in the translations as evidence of System F$^\circ$'s expressive power and its ability to handle unrestricted terms (the bulk of any most programs) in a concise way, justifying our claim that it provides linearity in a lightweight fashion.

3. Examples

To demonstrate System F$^\circ$'s applicability, we now turn to two categories of examples. First, we demonstrate that, while System F$^\circ$ does not build in notions of references and aliasing, protocols defining correct memory management can indeed be enforced by System F$^\circ$ types. Second, we prove that *any* protocol expressible as a finite automaton has a corresponding F$^\circ$ type; in addition to establishing that a rather large class of protocols can be encoded in our type system, this proof also highlights intuitions about linearity that still need formalization, which we will tackle in Section 4. (It is easy to see, however, that the regular languages are not an upper limit on System F$^\circ$'s expressivity; the classic non-regular parenthesis matching example has an obvious protocol type.)

3.1 Reference cells

The filesystem interface in Section 1.2 can, under an appropriate renaming and abstraction over the contents type, also be seen as an interface for linear reference cells:[6]

$$\begin{array}{rcl} \mathsf{Ref}[\tau](\alpha) & = & \{\ \mathsf{set} : \tau \xrightarrow{\star} \alpha \xrightarrow{\star} \alpha, \\ & & \ \ \mathsf{get} : \alpha \xrightarrow{\star} (\tau, \alpha), \\ & & \ \ \mathsf{free} : \alpha \xrightarrow{\star} \mathsf{Unit}\ \} \end{array}$$

$$\mathsf{mkRef} \ : \ \forall\beta{:}\star.\, \beta \xrightarrow{\star} \exists\alpha{:}\circ.\, (\alpha, \mathsf{Ref}[\beta](\alpha))$$

[6] Here and elsewhere, we separate the "type parameters" like τ from the linear "state parameters" like α using the notation $[\tau](\alpha)$.

On its own this is not particularly interesting, as a linear reference cell simply encodes the practice of threading a value through a program. Indeed, as with Haskell's State monad, we could instantiate mkRef such that α *is* τ. Instead, however, let us consider variations on Ref that make more sense as safe interfaces wrapping true, potentially unsafe reference cells, much as File in Section 1.2 protected the primitive, unchecked filehandle calls.

While the above Ref could be such a safe interface, a more obvious one—which itself requires no linearity—is the type GCRef:

$$\mathsf{GCRef}[\tau](\alpha) \;=\; \{\, \mathsf{set} : \tau \xrightarrow{\star} \alpha \xrightarrow{\star} \alpha,\; \mathsf{get} : \alpha \xrightarrow{\star} (\tau, \alpha) \,\}$$

$$\mathsf{mkGCRef} \;:\; \forall\beta{:}\star.\; \beta \xrightarrow{\star} \exists\alpha{:}\star.\; (\alpha, \mathsf{GCRef}[\beta](\alpha))$$

The operations given by $\mathsf{GCRef}[\tau](\alpha)$ are, of course, those of a garbage-collected reference cell of type τ. By hiding the reference type behind α we ensure that such garbage-collected references cannot be freed, and, in this case, α can be unrestricted.

In System F$^\circ$, however, we can also define references that begin their lives as linear (and manually managed) but later are put under the garbage collector's control. $\mathsf{Ref}[\tau](\alpha)$ simply needs an additional function to serve as the appropriate coercion:

$$\mathsf{gc} : \alpha \xrightarrow{\star} \exists\beta{:}\star.\; (\beta, \mathsf{GCRef}[\tau](\beta))$$

By consuming and not returning α, gc prevents free from being called on the now garbage-collected (but unrestricted) reference. We thus have a coercion from alias-free to potentially aliased pointers, a fact that, had we conflated linearity with alias-freedom, would run counter to our subkinding relation of $\star \leqslant \circ$. (Whether gc needs to do any work at run time depends on the implementation of the memory management system.)

We can take other approaches to memory management as well. For instance, an intermediate point between a strictly linear and a garbage collected reference is a reference that must be explicitly aliased, and where aliases must be explicitly discarded. We can define this easily enough:

$$\mathsf{RCRef}[\tau](\alpha) \;=\; \{\, \mathsf{set} : \tau \xrightarrow{\star} \alpha \xrightarrow{\star} \alpha,\; \mathsf{get} : \alpha \xrightarrow{\star} (\tau, \alpha),\; \mathsf{alias} : \alpha \xrightarrow{\star} (\alpha, \alpha),\; \mathsf{drop} : \alpha \xrightarrow{\star} \mathsf{Unit} \,\}$$

$$\mathsf{mkRCRef} \;:\; \forall\beta{:}\star.\; \beta \xrightarrow{\star} \exists\alpha{:}\circ.\; (\alpha, \mathsf{RCRef}[\beta](\alpha))$$

A straightforward implementation of mkRCRef could return a pair of the desired reference cell and an additional cell to act as a counter, along with alias and drop operations that adjust this counter. Both the primary cell and this counter could safely be freed when the count reaches zero.

However, while our access capability is still linear with RCRef, we can never be certain that we possess the only reference to cell in question. To remedy this, as is often done in capability calculi [2, 9], we can give our cells both an exclusive capability α and a shared capability β, with possession of the exclusive capability implying that no outstanding copies of the shared capability remain. If, for example, the cell contents should not be altered if any aliases exist, we can use

$$\begin{aligned}\mathsf{ShareRef}[\tau](\alpha,\beta) \;=\; \{\, & \mathsf{set} : \tau \xrightarrow{\star} \alpha \xrightarrow{\star} \alpha,\\ & \mathsf{getE} : \alpha \xrightarrow{\star} (\tau, \alpha), \quad \mathsf{getS} : \beta \xrightarrow{\star} (\tau, \beta),\\ & \mathsf{share} : \alpha \xrightarrow{\star} \beta, \quad\;\; \mathsf{claim} : \beta \xrightarrow{\star} \alpha \oplus \beta,\\ & \mathsf{alias} : \beta \xrightarrow{\star} (\beta, \beta), \quad \mathsf{drop} : \beta \xrightarrow{\star} \mathsf{Unit},\\ & \mathsf{free} : \alpha \xrightarrow{\star} \mathsf{Unit} \,\}\end{aligned}$$

$$\mathsf{mkShareRef} \;:\; \forall\gamma{:}\star.\; \gamma \xrightarrow{\star} \exists\alpha{:}\circ.\; \exists\beta{:}\circ.\; (\alpha, \mathsf{ShareRef}[\gamma](\alpha,\beta))$$

Here $\alpha \oplus \beta$ is a standard sum type; as mentioned in Section 2, these are not encodable in System F$^\circ$ as presented so far, but they are an easy extension and are discussed in Section 5.1. The claim function exchanges a shared β for an exclusive α when the underlying counter indicates that only one alias exists; in all other cases it simply returns the supplied shared capability.

We can do still more if we extend System F$^\circ$ with quantification over higher kinds, an extension which meshes well with Section 5.1's datatypes. For instance, we can define linear references that support *strong updates*—that is, updates that change the type contained in the reference cell—by abstracting the type of the access capability over the type of the contents, giving it kind $\star \Rightarrow \circ$:

$$\begin{aligned}\mathsf{SURef}[\tau](\alpha) \;=\; \{\, & \mathsf{set} : \forall\beta{:}\star.\; \forall\gamma{:}\star.\; \gamma \xrightarrow{\star} \alpha\; \beta \xrightarrow{\star} \alpha\; \gamma,\\ & \mathsf{get} : \forall\beta{:}\star.\; \alpha\; \beta \xrightarrow{\star} (\beta, \alpha\; \beta),\\ & \mathsf{free} : \forall\beta{:}\star.\; \alpha\; \beta \xrightarrow{\star} \mathsf{Unit} \,\}\end{aligned}$$

$$\mathsf{mkSURef} \;:\; \forall\beta{:}\star.\; \beta \xrightarrow{\star} \exists\alpha{:}\star \Rightarrow \circ.\; (\alpha\; \beta, \mathsf{SURef}[\beta](\alpha))$$

Operation records of type $\mathsf{SURef}[\tau](\alpha)$ can easily be coerced to type $\mathsf{Ref}[\tau](\alpha\; \tau)$ by partial application of member functions. Augmenting SURef along the lines of ShareRef could further allow for sharable reference cells that support strong updates only when they are not shared, a fairly sophisticated feature.

3.2 Regular protocols

Apart from memory cells and file operations, what other protocols can System F$^\circ$ enforce? Rather than present more individual examples, we will show how *any* protocol expressible as a regular language can be written in F$^\circ$.

We take the standard definition of a DFA as a tuple $M = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is a finite set of alphabet symbols (in our context, protocol actions), δ is a subset of $\Sigma \times Q \times Q$ (that is, a ternary relation among actions, current states, and next states), q_0 is a distinguished initial state, and F is a set of final states. The file access protocol from Section 1.2, for instance, can be thought of as a very simple automaton with states Open and Closed and an alphabet consisting of read, write, and close, with open as the creator of such an automaton.

For ease of notation, we extend our metavariable conventions to allow q and r as type variables. Taking $Q = \{q_0, \ldots, q_n\}$, we can now define the automaton type for the DFA M as

$$\begin{aligned}\tau_M \;=\; \exists q_0{:}\circ, \ldots, q_n{:}\circ.\; (\; & q_0,\\ & \{\; a\_\mathsf{takes}\_q\_\mathsf{to}\_q' : q \xrightarrow{\star} q'\\ & \qquad \textit{for every } (a, q, q') \in \delta\\ & \quad \vdots\\ & \quad \mathsf{done\_at}\_q : q \xrightarrow{\star} \mathsf{Unit}\\ & \qquad \textit{for every } q \in F \;\} \;)\end{aligned}$$

Such type is trivially inhabited: one can supply an existential package of this type where all type variables are bound to Unit.

We say that the evaluation to a value of an expression e containing a subexpression of type τ_M *reflects* a word $w = a_0 \ldots a_k$ if the sequence of record fields used is exactly

$$a_0\_\mathsf{takes}\_q_0\_\mathsf{to}\_r_0, \ldots a_k\_\mathsf{takes}\_r_{k-1}\_\mathsf{to}\_r_k$$

for some states r_0 through r_k. To prove that τ_M is an accurate representation of M, we need to show, first, that each $w = a_0 \ldots a_k$ accepted by M corresponds to an expression of type $\tau_M \xrightarrow{\star} \mathsf{Unit}$ that reflects w and, second, that each f of type $\tau_M \xrightarrow{\star} \mathsf{Unit}$ reflects some w accepted by M. The former is fairly straightforward:

Lemma 9. *For any $w = a_0 \ldots a_k$ accepted by M, there exists f such that $\cdot \vdash f : \tau_M \xrightarrow{\star} \mathsf{Unit}$ and for any $n : \tau_M$, $f\ n$ reflects w.*

Proof. Recall that, if $w = a_0 \dots a_k$ is accepted by M, we have a trace of M on w of the form $q_0 a_0 r_0 \dots r_{k-1} a_k r_k$, where $(a_0, q_0, r_0) \in \delta$, $(a_i, r_{i-1}, r_i) \in \delta$, and $r_k \in F$. Knowing this and the definition of τ_M, we can construct

$$\begin{aligned} f \;=\; & \lambda^\star n{:}\tau_M.\ \textbf{unpack}\ (q_0, \dots, q_n, p) = n\ \textbf{in} \\ & \quad \textbf{let}\ (\mathit{start}, \mathit{ops}) = p\ \textbf{in} \\ & \quad \textbf{let}\ s_0 = \mathit{ops}.a_0\_\mathsf{takes}\_q_0\_\mathsf{to}\_r_0\ \mathit{start}\ \textbf{in} \\ & \quad \vdots \\ & \quad \textbf{let}\ s_k = \mathit{ops}.a_k\_\mathsf{takes}\_r_{k-1}\_\mathsf{to}\_r_k\ s_{k-1}\ \textbf{in} \\ & \quad x.\mathsf{done\_at}\_r_k\ s_k \end{aligned}$$

This clearly reflects w, and, if w is indeed accepted by M, it will typecheck successfully. □

The other direction depends on arguments from parametricity (which we have already proved) and the nature of linearity (which we will formalize and prove in Section 4).

Lemma 10. *If* $\cdot \vdash f : \tau_M \xrightarrow{\star} \mathsf{Unit}$*, then there exists some* w *such that* M *accepts* w *and* f n *reflects* w *for any* n *where* $\cdot \vdash n : \tau_M$*.*

Proof. Because τ_M is linear, f must eliminate n before it can return anything of type Unit. This will require first unpacking the existential and then pattern-matching against the pair, leaving the linear *start* and unrestricted *ops*.

If $q_0 \in F$, then f might immediately apply $\mathit{ops}.\mathsf{done\_at}\_q_0$. This reflects the empty word ϵ, and, indeed, if $q_0 \in F$, then M accepts ϵ. Alternatively, regardless of whether $q_0 \in F$, *start* can become some other state type (or even q_0 again) by repeated applications of the $\mathit{ops}.a\_\mathsf{takes}\_q\_\mathsf{to}\_q'$ functions. If, after such applications, the result can be eliminated by some $\mathit{ops}.\mathsf{done\_at}\_q_j$, then it must be that $q_j \in F$. Moreover, because of the construction of τ_M, each $\mathit{ops}.a\_\mathsf{takes}\_q\_\mathsf{to}\_q'$ application must represent a valid transition of δ. Thus we are reflecting some w accepted by M. □

In the above proof, familiar intuitions about parametricity justify the arguments that f behaves the same way for any argument of type τ_M and that expressions whose types are variables can only be used in certain ways. The assurance what w is properly reflected, however, also depends on the connection between the static property of linear typing and the behavior of expressions at runtime. We would like type soundness to guarantee that our intuitions about the behavior of linear expressions are valid; in the next section we prove that this is indeed the case.

4. Linear Semantics

The preceding section makes clear that we have yet to formalize all of our intuitions about what linearity means for run-time behavior. This is not as straightforward as it might appear, because some of our intuitions about linearity turn out to be misleading. In essence, this is because linearity restricts *variables* of linear type, while we want to reason, at runtime, about the behavior of *expressions*.

For instance, linearity does not guarantee that subexpressions—or even values—of linear type will be used exactly once. Nor would we want it to: recall that the encoding of **let** expressions in Figure 4 involves a linear function, and we certainly want to allow for **let** expressions of unrestricted type.

However, there are examples which, in a call-by-name or call-by-need setting, do behave in ways we'd like to prevent. For example, if we have unpacked a reference cell from Section 3.1 as (r, ops), we can dispose of it and continue with the expression e via

$$(\lambda^\star \_{:}\mathsf{Unit}.\ e)\ (\mathit{ops}.\mathsf{free}\ r)$$

But this call to free will only be evaluated in call-by-value—since its result is not used, other evaluation strategies would simply discard the subexpression, leaking memory, even though this is precisely the sort of error we hoped to rule out. Similar concerns for uniqueness types are known in the Clean community [28].

Call-by-value, with its assurance that function arguments are evaluated exactly once, seems well-suited to avoiding these problems, and indeed, in a call-by-value setting—with the addition of the value restriction—we are not able to construct such problematic examples. In other words, the restrictions already in place to ensure soundness in Section 2.1 are everything we need to support our intuitions about linearity. To prove that this is so, the remainder of this section provides an extended operational semantics with which we can reason about linearity at runtime.

$$[\text{L-AppLam}]\ (\lambda^\kappa x{:}\tau.\ e)\ v \xrightarrow[\epsilon]{(v:\tau)}\!\!\!\!\longrightarrow \{x \mapsto (v{:}\tau)\}e$$

$$[\text{L-TAppLam}]\ \Lambda\alpha{:}\kappa.\ v\ [\tau] \xrightarrow[\epsilon]{\epsilon}\!\!\!\!\longrightarrow \{\alpha \mapsto (\tau{:}\kappa)\}v$$

$$[\text{L-Tag}]\ (v{:}\tau) \xrightarrow[(v:\tau)]{\epsilon}\!\!\!\!\longrightarrow v \qquad [\text{L-TApp}]\ \frac{e \xrightarrow[D]{C}\!\!\!\!\longrightarrow e'}{e\ [\tau] \xrightarrow[D]{C}\!\!\!\!\longrightarrow e'\ [\tau]}$$

$$[\text{L-App1}]\ \frac{e_1 \xrightarrow[D]{C}\!\!\!\!\longrightarrow e_1'}{e_1\ e_2 \xrightarrow[D]{C}\!\!\!\!\longrightarrow e_1'\ e_2} \qquad [\text{L-App2}]\ \frac{e \xrightarrow[D]{C}\!\!\!\!\longrightarrow e'}{v\ e \xrightarrow[D]{C}\!\!\!\!\longrightarrow v\ e'}$$

$$[\text{K-TTag}]\ \frac{\Gamma \vdash \tau : \kappa}{\Gamma \vdash (\tau{:}\kappa) : \kappa} \qquad [\text{T-Tag}]\ \frac{\Gamma; \Delta \vdash v : \tau}{\Gamma; \Delta \vdash (v{:}\tau) : \tau}$$

$$[\text{T-TTag1}]\ \frac{\Gamma; \Delta \vdash e : \tau \quad \Gamma \vdash \tau : \kappa}{\Gamma; \Delta \vdash e : (\tau{:}\kappa)} \qquad [\text{T-TTag2}]\ \frac{\Gamma; \Delta \vdash e : (\tau{:}\kappa)}{\Gamma; \Delta \vdash e : \tau}$$

Figure 5. Linearity-aware semantics for linear System F

4.1 Annotated F°

What does it mean to "use" a subexpression? We might mean by this that an expression is evaluated to a value, that it is passed to a function, or that it is applied to an argument. In order to understand the run-time effects of linearity, however, we focus on the use of a value, which we divide into two phases:

1. A value is *conscripted* when it is substituted into a function body in place of a variable.
2. A previously conscripted value is *discharged* when it is used as a value to allow evaluation to continue—in a context where, were it replaced by a free variable, evaluation would be stuck. [7]

From this linearity's contribution is clear: in the course of evaluation every conscripted linear value not contained within the final result will be discharged exactly once. To formalize this, we define a linearity-aware operational semantics on expressions extended with tagged values (representing the arguments passed to functions) and tagged types (representing type arguments):

$$\begin{array}{rcl} \tau & ::= & \dots \mid (\tau{:}\kappa) \\ e & ::= & \dots \mid (v{:}\tau) \end{array}$$

$$C, D \quad ::= \quad \epsilon \mid (v{:}\tau) \mid C, C$$

Our new semantics is given in Figure 5: we write $e \xrightarrow[D]{C}\!\!\!\!\longrightarrow e'$ if e steps to e' while conscripting the sequence of tagged values C

[7] As variables are not values, a free variable on *either* side of an application results in a stuck expression by the semantics in Figure 2.

and discharging the sequence D. Values are conscripted in rule L-APPLAM, and, in a slight but critical departure from the usual operational semantics, it is the argument value tagged with the type expected by the function that is substituted into the function body. Similarly, while L-TAPPLAM does not conscript, it does tag its type argument with its expected kind, thus keeping a record of the kind at which said argument will be considered within its body.

Since a tagged value $(v{:}\tau)$ is not itself a value, we have the rule L-TAG, which both discharges the value and removes its tag. Intuitively this means that the argument to a previous function application is either about to be used again—regardless of which side of an application it is on—or that it is being returned as the final result of the computation. Of course, discharging may be delayed for some time, as tagged values may linger beneath function closures. As with the presentation in Section 2, evaluation does not depend on typing—types and kinds are simply recorded—and, in particular, our new operational semantics does not treat expressions of non-linear type differently from those of linear type.

The kinding and typing rules in Figure 5 are straightforward; the ability to conclude $e : \tau$ from $e : (\tau{:}\kappa)$ regardless of κ makes kind annotations completely transparent, but the same is not true of type annotations. The soundness results in Section 2.1 carry over to the extended system without complication.

As an example, consider an application of the linear polymorphic identity function $\Lambda\alpha{:}\circ.\ \lambda^{\circ}x{:}\alpha.\ x$ to the natural number 42 of type Nat. This expression steps as follows:

$$\begin{array}{rll} & (\Lambda\alpha{:}\circ.\ \lambda^{\circ}x{:}\alpha.\ x)\ [\mathsf{Nat}]\ 42 & \\ \xrightarrow{\epsilon}_{\epsilon} & (\lambda^{\circ}x{:}(\mathsf{Nat}{:}\circ).\ x)\ 42 & \text{L-TAPPLAM} \\ \xrightarrow{(42:(\mathsf{Nat}:\circ))}_{\epsilon} & (42{:}(\mathsf{Nat}{:}\circ)) & \text{L-APPLAM} \\ \xrightarrow{\epsilon}_{(42:(\mathsf{Nat}:\circ))} & 42 & \text{L-TAG} \end{array}$$

Or, taking the obvious tag-concatenating closure $\xrightarrow{C}_{D}{}^{*}$:

$$(\Lambda\alpha{:}\circ.\ \lambda^{\circ}x{:}\alpha.\ x)\ [\mathsf{Nat}]\ 42 \xrightarrow{(42:(\mathsf{Nat}:\circ))}_{(42:(\mathsf{Nat}:\circ))}{}^{*}\ 42$$

In other words, over the course of evaluation, the value 42 was passed to a function and used (in this case, returned as the final result) exactly once, and it was considered at type $(\mathsf{Nat}{:}\circ)$. Were it to appear more than once in the lower annotation, we would know that linearity had failed us—although 42 is of type Nat, here it is being considered as a linear type because α was declared to be linear, so after being conscripted at this type it should be discharged exactly once. Were it to instead appear as $(42{:}\mathsf{Nat})$, however, there should be no such restrictions on its use, as we can indeed write functions from Nat to Nat which use their arguments more than once or ignore them altogether.

This semantics is similar in scope to heap semantics for linear functional languages [26], and indeed we could have taken this approach by tracking, instead of the traces C and D, a single heap H. Such a semantics would perform substitution at application only if the argument type is unrestricted; for linear types, we would simply associate the variable with its argument in the heap, removing it when the variable is used. But, while we could still prove Theorem 12 with such a semantics, we need the ability to track unrestricted arguments—typically not placed in the heap—and to reason about the order of function calls in order to support the reasoning we used in Section 3, which we will return to in Section 4.3.

4.2 Linearity at run time

Before we can formalize our intuitions about linearity, we need some auxiliary definitions. We write $\mathfrak{T}(e)$ for the multiset of tagged values appearing within e, defined simply as:

$$\begin{array}{rclcrcl} \mathfrak{T}(x) & = & \emptyset & & \mathfrak{T}(\lambda^{\kappa}x{:}\sigma_1.\ e) & = & \mathfrak{T}(e) \\ \mathfrak{T}((v{:}\tau)) & = & \{(v{:}\tau)\} \uplus \mathfrak{T}(v) & & \mathfrak{T}(\Lambda\alpha{:}\kappa.\ v) & = & \mathfrak{T}(v) \\ \mathfrak{T}(e_1\ e_2) & = & \mathfrak{T}(e_1) \uplus \mathfrak{T}(e_2) & & \mathfrak{T}(e\ [\tau]) & = & \mathfrak{T}(e) \end{array}$$

We write $\{C\}$ to denote the treatment of a sequence as a multiset, and, if S is a multiset of tagged values, we write $S\backslash\kappa$ for the subset of S that omits any $(v{:}\tau)$ where $\cdot \vdash \tau : \kappa$.

We define *proper* expressions as those in which no unrestricted value contains a value as a subexpression tagged with linear type—that is, a well-typed expression e is proper iff, for every value subexpression v in e, if v has some type τ and τ can be given kind $\star$, then $\mathfrak{T}(v)\backslash\star = \emptyset$. This property is preserved by evaluation:

Lemma 11 (Proper expressions)**.** *If e is proper and $e \xrightarrow{C}_{D} e'$, then e' is also proper.*

Proof. Values can only become tagged in a subexpression by function application. Because of the value restriction, in order for a linear value to be substituted into an unrestricted value, a linear variable bound at an outer scope would need to appear under an unrestricted λ. Rule T-LAM forbids this. □

Thus, as long as we write our source expressions in the language described in Section 2, we will never evaluate to an improper result—this is essentially the runtime version of Lemma 5. If we add new constructs, this property will continue to hold as long as

1. constructs under which evaluation can proceed (like conventional tuples) are given unrestricted types only if all of their components are also unrestricted, and
2. constructs that suspend computation either contain only values (like Λ) or only typecheck at an unrestricted type given an empty linear context (like λ).

(Of course, it also suffices to require that a new construct always be typed linearly, as we do for additive conjunction in Section 5.1.)

We are now equipped to prove our main result, that linearity guarantees a correspondence between values conscripted and discharged at linear type:

Theorem 12 (Run time linearity)**.** *If e is proper and $e \xrightarrow{C}_{D} e'$, then $\mathfrak{T}(e)\backslash\star \uplus \{C\}\backslash\star = \mathfrak{T}(e')\backslash\star \uplus \{D\}\backslash\star$.*

Proof. By straightforward induction over the annotated evaluation relation; L-APPLAM is the only interesting case. From T-UVAR and Lemma 11 we know that, unless the argument occurs exactly once in the function body, it will be of unrestricted type and contain no linear tags. We thus preserve our tag balance. □

In other words, at runtime, arguments treated linearly are never duplicated or discarded, exactly as we would hope. We can also immediately prove the following corollary, summarizing everything we know about expressions with unrestricted type:

Corollary 13 (Unrestricted results)**.** *If e is proper, $\cdot \vdash e : \tau$, and $\cdot \vdash \tau : \star$, then there exists some v such that $e \xrightarrow{C}_{D}{}^{*} v$, $\mathfrak{T}(e)\backslash\star \uplus \{C\}\backslash\star = \{D\}\backslash\star$, and $\mathfrak{T}(v)\backslash\star = \emptyset$.*

Proof. Follows directly from soundness, strong normalization, Lemma 11, and Theorem 12. □

4.3 Applications

To demonstrate how the above applies to the DFA encoding given in Section 3.2, we first examine the simpler encodings of products and existential types from Section 2.

Products Our traces treat pairs exactly as we would expect. Given $e_1 \xrightarrow{C_1}_{D_1}{}^{*} v_1$ and $e_2 \xrightarrow{C_2}_{D_2}{}^{*} v_2$, where e_1 has type τ_1 and e_2 has type τ_2, recall that

$$\begin{array}{rcl} (e_1, e_2) & = & (\Lambda\alpha{:}\circ.\ \Lambda\beta{:}\circ.\ \lambda^{\star}x{:}\alpha.\ \lambda^{\circ}y{:}\beta.\ \Lambda\gamma{:}\circ. \\ & & \quad \lambda^{\circ}f{:}\alpha \xrightarrow{\circ} \beta \xrightarrow{\circ} \gamma.\ f\ x\ y)\ [\tau_1]\ [\tau_2]\ e_1\ e_2 \end{array}$$

$$
\begin{array}{rl}
e_2 & \xrightarrow[D]{C}{}^{*} \quad (\lambda^{\circ} x{:}\{\alpha \mapsto \tau\}\tau_1.\ \Lambda\beta{:}\circ.\ \lambda^{\kappa_1} f{:}(\forall\alpha{:}\kappa.\ \tau_1 \xrightarrow{\circ} \beta).\ f\ [\tau]\ x)\ v_1 \\
& \xrightarrow[\epsilon]{(v_1:\{\alpha \mapsto \tau\}\tau_1)} \quad \Lambda\beta{:}\circ.\ \lambda^{\kappa_1} f{:}(\forall\alpha{:}\kappa.\ \tau_1 \xrightarrow{\circ} \beta).\ f\ [\tau]\ (v_1{:}\{\alpha \mapsto \tau\}\tau_1) \\
\\
e_3 & \xrightarrow[\epsilon]{\epsilon} \quad (\lambda^{\kappa_1} f{:}(\forall\alpha{:}\kappa.\ \tau_1 \xrightarrow{\circ} (\tau'{:}\circ)).\ f\ [\tau]\ (v_1{:}\{\alpha \mapsto \tau\}\tau_1))\ (\Lambda\alpha{:}\kappa.\ \lambda^{\circ} x{:}\tau_1.\ e') \\
& \xrightarrow[(\Lambda\alpha:\kappa.\ \lambda^{\circ} x:\tau_1.\ e'{:}\forall\alpha{:}\kappa.\ \tau_1 \xrightarrow{\circ} (\tau':\circ))]{(\Lambda\alpha:\kappa.\ \lambda^{\circ} x:\tau_1.\ e'{:}\forall\alpha{:}\kappa.\ \tau_1 \xrightarrow{\circ} (\tau':\circ))}{}^{*} \quad (\Lambda\alpha{:}\kappa.\ \lambda^{\circ} x{:}\tau_1.\ e')\ [\tau]\ (v_1{:}\{\alpha \mapsto \tau\}\tau_1) \\
& \xrightarrow[\epsilon]{\epsilon} \quad (\lambda^{\circ} x{:}\{\alpha \mapsto (\tau{:}\kappa)\}\tau_1.\ \{\alpha \mapsto (\tau{:}\kappa)\}e')\ (v_1{:}\{\alpha \mapsto \tau\}\tau_1) \\
& \xrightarrow[(v_1:\{\alpha \mapsto \tau\}\tau_1)]{\epsilon} \quad (\lambda^{\circ} x{:}\{\alpha \mapsto (\tau{:}\kappa)\}\tau_1.\ \{\alpha \mapsto (\tau{:}\kappa)\}e')\ v_1 \\
& \xrightarrow[(v_1:\{\alpha \mapsto (\tau:\kappa)\}\tau_1)]{\epsilon} \quad \{x \mapsto (v_1{:}\{\alpha \mapsto (\tau{:}\kappa)\}\tau_1)\}e'
\end{array}
$$

Figure 6. Evaluation of annotated existentials

Clearly, then,

$$(e_1, e_2)\ \xrightarrow[D_1,\, D_2]{C_1,\, C_2,\, (v_1:(\tau_1:\circ)),\, (v_2:(\tau_2:\circ))}{}^{*}\ (v_1, v_2)$$

Given our representation of pairs as closures, this is reasonable; though the conscription trace tags v_1 and v_2 with linear types, they will be discharged at these types as soon as (v_1, v_2) is destructed and, assuming the provided function f is of declared type $\tau_1 \xrightarrow{\circ} \tau_2 \xrightarrow{\circ} \gamma$, reconscripted at their original, untagged types τ_1 and τ_2.

Existentials Existential types are a bit more complicated. Given $e_1 \xrightarrow[D]{C}{}^{*} v_1$ where e_1 has type $\{\alpha \mapsto \tau\}\tau_1$, τ_1 has kind κ_1, and τ has kind κ, let

$$
\begin{array}{rcl}
e_2 & = & \textbf{pack}\ \alpha{:}\kappa = \tau\ \textbf{in}\ e_1 : \tau_1 \\
& = & \textbf{let}\ x = e_1\ \textbf{in}\ \Lambda\beta{:}\circ.\ \lambda^{\kappa_1} f{:}(\forall\alpha{:}\kappa.\ \tau_1 \xrightarrow{\circ} \beta).\ f\ [\tau]\ x \\
& = & (\lambda^{\circ} x{:}\{\alpha \mapsto \tau\}\tau_1.\ \Lambda\beta{:}\circ. \\
& & \quad \lambda^{\kappa_1} f{:}(\forall\alpha{:}\kappa.\ \tau_1 \xrightarrow{\circ} \beta).\ f\ [\tau]\ x)\ e_1
\end{array}
$$

Letting v_2 refer to the result of fully evaluating e_2, we have $e_2\ \xrightarrow[D]{C,\, (v_1:\{\alpha \mapsto \tau\}\tau_1)}{}^{*}\ v_2$; the details of this reduction can be seen in Figure 6. This seems sensible for the encoding of an existential package, as it reflects exactly that the result of evaluating e_1 has been captured in a function closure, just as with the capture that occurs when applying the product constructor.

Now, to use v_2, take e' of type τ' and let

$$
\begin{array}{rcl}
e_3 & = & \textbf{unpack}\ \alpha, x = v_2\ \textbf{in}\ e' \\
& = & v_2\ [\tau']\ (\Lambda\alpha{:}\kappa.\ \lambda^{\circ} x{:}\tau_1.\ e')
\end{array}
$$

The evaluation of this expression, also shown in Figure 6, is a bit lengthier; the interesting thing to note, though, is that, before the existentially wrapped v_1 is passed to the function wrapping e', it is discharged at type $\{\alpha \mapsto \tau\}\tau_1$ and reconscripted at the type $\{\alpha \mapsto (\tau{:}\kappa)\}\tau_1$. This stricter reconscription stands in direct contrast to what occurs in the elimination of pairs, and, if κ is $\circ$, we know that any component of v_1 with type $(\tau{:}\kappa)$ must be discharged exactly once in e' for each time it is conscripted, regardless of whether or not τ can also be given kind $\star$.

Subsequently, of course, it could be reconscripted simply at type τ, but thanks to parametricity we know that e' cannot do this on its own—as in the filesystem example, it would need to make use of a function already present in the existential package.

DFAs Finally, to show how the arguments made in proving of Lemma 10 are made rigorous by this machinery, consider making two more minor syntactic changes to System F$^{\circ}$: first, rather than writing $\lambda^{\kappa} x{:}\tau.\ e$ for functions, write $\textbf{fn}^{\kappa}\ \mathsf{name}\ x{:}\tau.\ e$, and second, rather than the simple L-TAG, add rules for the left and right sides of applications which further add L and R tags to discharged values, along with a final result rule which adds no further tag. It doesn't matter what names we give to functions, but if we choose to use our record labels as function names when possible—and require that all other names are drawn from some set disjoint from said record labels—then our definition of an evaluation reflecting a word becomes very simple: the DFA trace must be visible in D by examining the values tagged with L.

We can now return to the proof given in Section 3 and consider the problem in terms of value traces. After unpacking the existential, we will have the linear initial state conscripted but not yet discharged, and because our entire expression has the unrestricted type Unit, we know from Corollary 13 that it must be discharged before the final result is returned. Parametricity further restricts states to being discharged on the right side of applications. Then, as before, it is simply a matter of following the state as functions are applied to it, noting the labels that can become attached to the value trace; in the end, when some $\mathsf{done\_at}\_q$ is applied, our value trace will contain a clear accepting DFA trace.

$$
\begin{array}{rcll}
\tau & ::= & \ldots \mid (\tau, \ldots, \tau) & \\
e & ::= & \ldots \mid (e, \ldots, e) \mid \textbf{let}\ (x_1, \ldots, x_n) = e\ \textbf{in}\ e & \\
v & ::= & \ldots \mid (v, \ldots, v) &
\end{array}
$$

$$[\text{K-PROD}]\ \frac{\Gamma \vdash \tau_1 : \kappa\ \ldots\ \Gamma \vdash \tau_n : \kappa}{\Gamma \vdash (\tau_1, \ldots, \tau_n) : \kappa}$$

$$[\text{T-PROD}]\ \frac{\Gamma; \Delta_1 \vdash e_1 : \tau_1\ \ldots\ \Gamma; \Delta_n \vdash e_n : \tau_n \quad \Delta_1 \Cup \ldots \Cup \Delta_n = \Delta}{\Gamma; \Delta \vdash (e_1, \ldots, e_n) : (\tau_1, \ldots, \tau_n)}$$

$$[\text{T-PLET}]\ \frac{\Gamma; \Delta_1 \vdash e_1 : (\tau_1, \ldots, \tau_n) \quad [\Gamma; \Delta_2], x_1 : \tau_1, \ldots, x_n : \tau_n \ni \Gamma', \Delta_2' \quad \Gamma'; \Delta_2' \vdash e_2 : \tau \quad \Delta_1 \Cup \Delta_2 = \Delta}{\Gamma; \Delta \vdash \textbf{let}\ (x_1, \ldots, x_n) = e_1\ \textbf{in}\ e_2 : \tau}$$

Figure 7. Syntax and static rules for multiplicative products

5. Extensions: Towards Practicality

Although System F$^{\circ}$ as presented so far can serve as an expressive core calculus, to be usable in practice it is necessary to extend it with features suitable for a surface language. This section shows how to add a variety of useful constructs familiar from functional programming. None of these extensions are particularly difficult, though we point out a few places where linearity must be minded.

5.1 Language additions

Polykinded products System F$^{\circ}$ as it stands can already encode both unrestricted and linear (*multiplicative*) products. In practice, however, it is useful to build in support for n-ary tuples. As shown in Figure 7, the typing rules are mostly standard; evaluation rules are completely standard and have been omitted. The kinding rule K-PROD, in combination with subkinding, allows the same syntax

$$\begin{array}{rcl} \tau & ::= & \ldots \mid \langle \tau, \ldots, \tau \rangle \\ e & ::= & \ldots \mid \langle e, \ldots, e \rangle \mid e.i \\ v & ::= & \ldots \mid \langle e, \ldots, e \rangle \end{array} \qquad [\text{E-ADD}]\ \frac{e \longrightarrow e'}{e.i \longrightarrow e'.i}$$

$$[\text{E-CHOOSE}]\ \langle e_1, \ldots, e_i, \ldots, e_n \rangle.i \longrightarrow e_i$$

$$[\text{K-ADD}]\ \frac{\Gamma \vdash \tau_1 : \circ \ \ldots\ \Gamma \vdash \tau_n : \circ}{\Gamma \vdash \langle \tau_1, \ldots, \tau_n \rangle : \circ}$$

$$[\text{T-ADD}]\ \frac{\Gamma; \Delta \vdash e_1 : \tau_1 \ \ldots\ \Gamma; \Delta \vdash e_n : \tau_n}{\Gamma; \Delta \vdash \langle e_1, \ldots, e_n \rangle : \langle \tau_1, \ldots, \tau_n \rangle}$$

$$[\text{T-SEL}]\ \frac{\Gamma; \Delta \vdash e : \langle \tau_1, \ldots, \tau_i, \ldots, \tau_n \rangle}{\Gamma; \Delta \vdash e.i : \tau_i}$$

Figure 8. Syntax and rules for the additive products

$$e \quad ::= \quad \ldots \mid \textbf{fix}\ f.\ v$$

$$[\text{E-FIX}]\ \textbf{fix}\ f.\ v \longrightarrow \{f \mapsto (\textbf{fix}\ f.\ v)\}v$$

$$[\text{T-FIX}]\ \frac{\Gamma, f{:}\tau; \cdot \vdash v : \tau \qquad \Gamma \vdash \tau : \star}{\Gamma; \cdot \vdash \textbf{fix}\ f.\ v : \tau}$$

Figure 9. Syntax and rules for fixpoints.

to be used for both linear and unrestricted tuples; if any component type of a tuple is linear, so is the tuple itself. T-PLET uses an n-ary version of the nondeterministic bind operation (recall T-LAM), allowing unrestricted components of a tuple to be used arbitrarily but binding linear components in the linear context.

Additive products As mentioned in Section 2, we cannot encode linear sums in System F$^\circ$ as presented so far—we have no means of sharing the linear context among different subexpressions, all but one of which will be dropped. Connectives that enable this are referred to as *additive* in the terminology of linear logic; traditionally they are written as $\tau_1 \oplus \tau_2$ for sums, and $\tau_1 \,\&\, \tau_2$ for *additive products*—lazy linear products eliminated by projection rather than pattern matching. In keeping with our syntactic conventions, however, we introduce n-ary additive products $\langle \tau_1, \ldots, \tau_n \rangle$; sums could be encoded in terms of these products but are also subsumed by the algebraic datatypes discussed below.

Figure 8 gives the new syntax, definitions, and static and dynamic rules for n-ary additive products. Note that additive products necessarily denote suspended computations, as further evaluation progress cannot be made until a branch is chosen. Unlike multiplicative products, additive products always have kind $\circ$ regardless of the kinds of the component types, as seen in K-ADD.

Fixpoints Adding support for fixpoint computation turns out to be remarkably straightforward: recursion implies the ability to repeatedly invoke a function within its own body, which naturally puts us in the unrestricted portion of System F$^\circ$. Figure 9 shows the new syntax and rules, where we write **fix** $f.\ v$ for the computation that immediately unwinds itself to a value as shown in E-FIX. Since these operational semantics might duplicate v, T-FIX requires that fixpoint expressions be closed with respect to the linear context; note that this is compatible with Lemma 4 (3).

Algebraic datatypes Figure 10 shows the static rules for general recursive, polymorphic datatypes—as with multiplicative products, the dynamic rules are standard. We assume a signature Σ_T that maps a datatype constructor T to its (higher) kind $\overline{\kappa}$, which describes T's type parameters; T has arity k_T. Term-level constructors for type T are written ctr_i^T, and their types are given by the map Σ_c. The type of a term-level constructor is polymorphic over the parameters of the datatype, but it may also expect additional type parameters, which act as through they are existentially bound—this design is similar to that found in Haskell. Importantly, if the result kind of the type constructor T is $\star$, then the value parameters to the term-level constructors must also be of kind $\star$. This constraint, written as part of the signature well formedness checks in Figure 10, is necessary to ensure that an unrestricted datatype can't hide a linear value and thus allow it to be duplicated or discarded.

The additive nature of datatypes is shown in T-CASE, which uses the same linear context Δ_2 when checking each branch. The branches themselves are polymorphic linear functions—additionally abstracted over any existential type parameters—one of which will be applied to create a result of the appropriate type. As with additive products, each branch will capture the same linear free variables, but only one of the branches will fire at run time.

5.2 Syntactic support for linearity

In addition to making it possible for types to express safe protocols over stateful resources, it must also be convenient for programmers to write client code that interacts with these interfaces. For instance, a file copy program, using the interface given in Section 1.2, should look like simple imperative code and not require the programmer to think explicitly about existentially bound linear type variables and the threading of linear filehandles. In the rest of this section we sketch out some possible syntactic support—partly inspired by but simpler than Haskell's type classes—for programming with linearity, which allows us to copy files as follows[8]:

```
let copy = fix f.
  action ([File] h1, [File] h2).
    y ← h1.read;
    if y = EOF then return ()
    else h2.write y;
         (h1, h2)..f
in
start [File] h1 = open "InFile.txt";
start [File] h2 = open "OutFile.txt";
(h1, h2)..copy;
h1.close; h2.close
```

This program would fail to typecheck if either of the calls to close were omitted, if one of the file handles were closed within the body of copy, or if a read or write operation were called after close. In the rest of this section we will show how to translate from this syntax into System F$^\circ$ as described up to this point.

First, we make more explicit the pattern common to the protocol examples we have seen thus far. Let $\Theta = \alpha_1 \ldots \alpha_n$ be a list of type variables representing the states of the protocol.[9] The possible actions of the protocol can be given by a record of operations, each of which causes a state transition from σ_{in} to σ_{out}.

$$\begin{array}{rl} \mathsf{Ops}(\Theta) = \{ & \mathsf{op}_1 : \forall \overline{\beta_1{:}\kappa_1}.\tau_{in_1} \xrightarrow{\star} \sigma_{in_1} \xrightarrow{\star} (\tau_{out_1}, \sigma_{out_1}) \\ & \ldots \\ & \mathsf{op}_m : \forall \overline{\beta_m{:}\kappa_m}.\tau_{in_m} \xrightarrow{\star} \sigma_{in_m} \xrightarrow{\star} (\tau_{out_m}, \sigma_{out_m}) \ \} \end{array}$$

Here, $\sigma_{in_j} \in \Theta$ is the start state of the operation and $\sigma_{out_j} \in \Theta \cup \{\mathsf{Unit}\}$ is either the end state of the operation or Unit, indicating

[8] Technically, to match up with what follows, the return type of write must be (Unit, α) rather than just α; we also assume that EOF is of type Char.

[9] In this section, we write $\forall\Theta{:}\circ.\tau$ as short hand for $\forall\alpha_1{:}\circ. \ldots \forall\alpha_n{:}\circ.\tau$, and similarly for existentials.

$$\begin{array}{lcl} \bar{\kappa} & ::= & \kappa \mid \kappa \Rightarrow \bar{\kappa} \\ \tau & ::= & \ldots \mid T\ \tau_1 \ldots \tau_n \end{array} \qquad \begin{array}{lcl} e & ::= & \ldots \mid \mathit{ctr}_i^T\ \tau_1 \ldots \tau_n\ e \mid \textbf{case}\ e\ \textbf{of}\ \mathit{ctr}_1^T.\ v_1 \mid \ldots \mid \mathit{ctr}_n^T.\ v_n \\ v & ::= & \ldots \mid \mathit{ctr}_i^T\ \tau_1 \ldots \tau_n\ v \end{array}$$

$$[\text{K-T}]\ \frac{\Sigma_T \vdash T : \kappa_1 \Rightarrow \ldots \kappa_{k_T} \Rightarrow \kappa_T \quad \Gamma \vdash \tau_1 : \kappa_1 \ldots \Gamma \vdash \tau_{k_T} : \kappa_{k_T}}{\Gamma \vdash T\ \tau_1 \ldots \tau_{k_T} : \kappa_T} \qquad [\text{T-Ctr}]\ \frac{\Sigma_c \vdash \mathit{ctr}_i^T : \forall \alpha_1{:}\kappa_1, \ldots, \alpha_n{:}\kappa_n.\tau \xrightarrow{\star} T\ \alpha_1 \ldots \alpha_{k_T} \quad \Gamma;\Delta \vdash e : \tau \quad \Gamma \vdash \tau_1 : \kappa_1\ \ldots\ \Gamma \vdash \tau_n : \kappa_n}{\Gamma;\Delta \vdash \mathit{ctr}_i^T\ \tau_1 \ldots \tau_{k_T}, \tau_{k_T+1}, \ldots, \tau_n\ e : T\ \tau_1 \ldots \tau_{k_T}}$$

$$[\text{T-Case}]\ \frac{\Delta_1 \Cup \Delta_2 = \Delta \quad \Gamma;\Delta_1 \vdash e : T\ \tau_1 \ldots \tau_{k_T} \quad \alpha_1 \ldots \alpha_n \notin \Gamma, \tau \quad \left(\begin{array}{c} \Sigma_c \vdash \mathit{ctr}_i^T : \forall \alpha_1{:}\kappa_1, \ldots, \alpha_{k_T}{:}\kappa_{k_T}, \alpha_{k_T+1}{:}\kappa_{k_T+1}, \ldots \alpha_{n_i}{:}\kappa_{n_i}.\tau_i \xrightarrow{\star} T\ \alpha_1 \ldots \alpha_{k_T} \\ \Gamma, \alpha_1{:}\kappa_1, \ldots, \alpha_{k_T}{:}\kappa_{k_T};\Delta_2 \vdash v_i : \forall \alpha_{k_T+1}{:}\kappa_{k_T+1}, \ldots \alpha_{n_i}{:}\kappa_{n_i}.\tau_i \xrightarrow{\circ} \tau \end{array} \right)^{(i \in 1 \ldots m)}}{\Gamma;\Delta \vdash \textbf{case}\ e\ \textbf{of}\ \mathit{ctr}_1^T.\ v_1 \mid \ldots \mid \mathit{ctr}_m^T.\ v_m : \tau} (T\ \mathit{has}\ m\ \mathit{constructors})$$

$\Sigma_T \vdash \Sigma_c$ holds if and only if for every T such that $\Sigma_T \vdash T : \kappa_1 \Rightarrow \ldots \kappa_{k_T} \Rightarrow \kappa_T$ and ctr_i^T such that $\Sigma_c \vdash \mathit{ctr}_i^T : \forall \alpha_1{:}\kappa_1, \ldots, \alpha_{n_i}{:}\kappa_{n_i}.\tau_i \xrightarrow{\star} T\ \alpha_1 \ldots \alpha_{k_T}$ it is the case that $\alpha_1{:}\kappa_1, \ldots \alpha_{n_i}{:}\kappa_{n_i} \vdash \tau_i : \kappa_T$

Figure 10. Syntax and static rules for polymorphic, recursive datatypes. Σ_T and Σ_c are global constructor contexts such that $\Sigma_T \vdash \Sigma_c$

that the protocol is complete. We assume that each operation might be polymorphic and, for simplicity, that each takes one input of type τ_{in_j} and produces an output of type τ_{out_j}, both of which might contain occurrences of variables in Θ. Almost all of the protocol examples we have seen can be written in this form.

If $\alpha_{init} \in \Theta$ is the start state, then an initial instance e_1 of the protocol is represented by a value of type $\exists \Theta{:}\circ.(\alpha_{init}, \mathsf{Ops}(\Theta))$. Thanks to α_{init}, this value has linear kind, so it must be unpacked and the resulting tuple destructed. We suggest a convenient notation to simplify this process:

$$\begin{array}{rcl} \textbf{start}\ [\mathsf{Ops}]\ h = e_1\ \textbf{in}\ e_2 & \triangleq & \textbf{unpack}\ \Theta_h, x = e_1\ \textbf{in} \\ & & \textbf{let}\ (h, \mathit{ops}_h) = x\ \textbf{in}\ e_2 \end{array}$$

Here, Θ_h is a list of fresh type variables, h is bound to the linear state, and ops_h is the record of operations associated with h. The syntax includes the annotation Ops to help with type checking and to determine what syntactic sugar applies with respect to h in the body e_2. Note that ops_h has type $\mathsf{Ops}(\Theta_h)$, which is unrestricted and hence may be duplicated at will; this invariant will be maintained throughout the interpretation of our syntax.

The intuition behind our scheme is simple: we use the single name h for the linear handle associated with the thread of a protocol—this implicit reuse can never be an issue, since the variable is linear. This handle (which must be typed by a variable in Θ_h) then determines an appropriate record of operations, ops_h, associated with the protocol. The programmer never mentions ops_h explicitly; rather it is threaded through the computation automatically, much like a typeclass dictionary.

The basic notation treats the protocol operations as "methods" of the linear handle h with implicit argument ops_h, binding the result to the variable y and conflating the initial and result handles:

$$y \leftarrow h.\mathit{op}_i\ [\overline{\tau}]\ e_1;\ e_2 \quad \triangleq \quad \textbf{let}\ (y, h) = \mathit{ops}_h.\mathit{op}_i\ [\overline{\tau}]\ e_1\ h\ \textbf{in}\ e_2$$

If the programmer wishes to write a client function that takes one or more handles, each following its own protocol, the function must be polymorphic over the appropriate *ops* records. We call such functions "actions"; it makes sense to give them a type very similar to the operations of a protocol, since they must take a vector of resource handles each in some protocol state and return a new vector of protocol states.

$$\begin{array}{l} \textbf{action}\ ([\mathsf{Ops}_1]\ h_1, \ldots, [\mathsf{Ops}_n]\ h_n)\ \overline{\beta{:}\kappa}\ (x{:}\tau_{in}).\ e \quad \triangleq \\ \quad \Lambda \Theta_{h_1}{:}\circ.\ \lambda^\star \mathit{ops}_{h_1}{:}\mathsf{Ops}_1(\Theta_{h_1}).\ \ldots \Lambda \Theta_{h_n}{:}\circ.\ \lambda^\star \mathit{ops}_{h_n}{:}\mathsf{Ops}_1(\Theta_{h_n}). \\ \quad\quad \Lambda \overline{\beta{:}\kappa}.\ \lambda^\star x{:}\tau_{in}.\ \lambda^\circ s{:}(\alpha_{h_1}, \ldots, \alpha_{h_n}). \\ \quad \textbf{let}\ (h_1, \ldots, h_n) = s\ \textbf{in}\ e \end{array}$$

Here e must have type $(\tau_{out}, (\sigma_{out_1}, \ldots, \sigma_{out_n}))$, where $\sigma_{out_j} \in \Theta_{h_j}$ and the type of h_j must be some α_{h_j} in Θ_{h_j}, representing the state that handle must be in when this function is called. Our syntactic sugar requires that type inference be able to determine the α_{h_j}'s by inspecting e. We expect this to be rather straightforward, since the type is uniquely determined by the first operation invoked on the handle; alternatively we could add a type annotation to h_j. Inside the body of such an action, we allow a convenient means of packaging the state values to be returned to the caller:

$$\textbf{return}\ e \quad \triangleq \quad (e, (h_1, \ldots, h_n))$$

Taken together, these constraints ensure that the type of an action is a polymorphic operation over a vector of states; *i.e.*, it has the type

$$\begin{array}{l} \forall \Theta_{h_1}{:}\circ.\ \mathsf{Ops}_1(\Theta_{h_1}) \xrightarrow{\star} \ldots \forall \Theta_{h_n}{:}\circ.\ \mathsf{Ops}_n(\Theta_{h_n}) \xrightarrow{\star} \\ \quad \forall \overline{\beta{:}\kappa}.\ \tau_{in} \xrightarrow{\star} (\alpha_{h_1}, \ldots, \alpha_{h_n}) \xrightarrow{\star} (\tau_{out}, (\sigma_{out_1}, \ldots, \sigma_{out_n})) \end{array}$$

Inside the body of an action, the "method call" syntax defined earlier can be used to invoke protocol operations on the handles h_j. Since an action is just another sort of operation over handles—albeit a higher level one—we would like similar syntax for invoking such an operation on a list of handles. Assuming f has the type of an action as given above, we define:

$$\begin{array}{l} y \leftarrow (h_1, \ldots, h_n)..f\ [\overline{\tau}]\ e_1;\ e_2 \triangleq \\ \quad \textbf{let}\ (y, s) = f[\Theta_{h_1}]\ \mathit{ops}_{h_1} \ldots [\Theta_{h_n}]\ \mathit{ops}_{h_n}[\overline{\tau}]\ e_1\ (h_1, \ldots, h_n)\ \textbf{in} \\ \quad \textbf{let}\ (h_1, \ldots, h_n) = s\ \textbf{in}\ e_2 \end{array}$$

One can easily define variants of both the above and the earlier protocol operation notation that omit the argument e_1 or the binding for y in the case that one or both of τ_{in} and τ_{out} are Unit, or that do not rebind the handle variable in cases where the handle is consumed, as with close. Also, if the operation invocation is last in a sequence, e_2 and the corresponding the let binding may be omitted, since the result of the operation is the result of the sequence—this is useful, *e.g.*, for making tail calls.

Of course, there are many ways of using linear types, and the above does not account for all of them. We believe, however, that it does show the feasibility of making System F$^\circ$ protocols palatable to the programmer, and that this provides further evidence that linearity as implemented in System F$^\circ$ deserves to be considered for inclusion in more mainstream functional programming languages.

6. Conclusion

We have presented System F$^\circ$, a simple variant of the linear polymorphic λ-calculus that nevertheless can enforce a rich variety of

protocols. System F$^\circ$ is sound and enjoys parametricity, and we have proved that protocol enforcement is faithful—that is, linear resources are never misused—thanks to these properties. We have demonstrated the applicability of System F$^\circ$ through a variety of examples and by showing how to extend it with features geared towards practical programming with linear types.

References

[1] Amal Ahmed, Matthew Fluet, and Greg Morrisett. A step-indexed model of substructural state. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 78–91, New York, NY, USA, 2005. ACM.

[2] Amal Ahmed, Matthew Fluet, and Greg Morrisett. L3: A linear language with locations. *Fundam. Inf.*, 77(4):397–449, 2007.

[3] Andrew Barber. *Linear Type Theories, Semantics and Action Calculi*. PhD thesis, Edinburgh University, 1997.

[4] Nick Benton, G. M. Bierman, J. Martin E. Hyland, and Valeria de Paiva. A term calculus for intuitionistic linear logic. In M. Bezem and J. F. Groote, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 75–90. Springer-Verlag LNCS 664, 1993.

[5] P. N. Benton. A mixed linear and non-linear logic: proofs, terms and models. In *Proceedings of Computer Science Logic (CSL '94), Kazimierz, Poland.*, pages 121–135. Springer-Verlag, 1995.

[6] G. M. Bierman, A. M. Pitts, and C. V. Russo. Operational properties of lily, a polymorphic linear lambda calculus with recursion. In *Fourth International Workshop on Higher Order Operational Techniques in Semantics, Montral, volume 41 of Electronic Notes in Theoretical Computer Science*. Elsevier, 2000.

[7] Gavin M. Bierman. Program equivalence in a linear functional language. *Journal of Functional Programming*, 10(2), 2000.

[8] Lars Birkedal, Rasmus Ejlers Møgelberg, and Rasmus Lerchedahl Petersen. Category-theoretic models of Linear Abadi & Plotkin Logic. *Theory and Applications in Categories*, 20(7), 2008.

[9] Arthur Charguéraud and François Pottier. Functional translation of a calculus of capabilities. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 213–224, New York, NY, USA, 2008. ACM.

[10] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 262–275, San Antonio, Texas, January 1999.

[11] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proc. of the SIGPLAN Conference on Programming Language Design*, pages 59–69, Snowbird, UT, June 2001.

[12] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in singularity os. *SIGOPS Oper. Syst. Rev.*, 40(4):177–190, 2006.

[13] Manuel Fähndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *Proc. of the SIGPLAN Conference on Programming Language Design*, pages 13–24, Berlin, Germany, June 2002.

[14] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arith mé tique d'ordre supérieur*. Thèse d'état, University of Paris VII, 1972. Summary in J. E. Fenstad, editor, *Scandinavian Logic Symposium*, pp. 63–92, North-Holland, 1971.

[15] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[16] Jean-Yves Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge University Press, 1989.

[17] Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. Experience with safe manual memory-management in Cyclone. In *ISMM '04: Proceedings of the 4th international symposium on Memory management*, pages 73–84, New York, NY, USA, 2004. ACM.

[18] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP98, volume 1381 of LNCS*, pages 122–138. Springer-Verlag, 1998.

[19] Atsushi Igarashi and Naoki Kobayashi. Resource usage analysis. *ACM Trans. Program. Lang. Syst.*, 27(2):264–313, 2005.

[20] Oleg Kiselyov and Chung-chieh Shan. Lightweight monadic regions. In *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*, pages 1–12, New York, NY, USA, 2008. ACM.

[21] Yves Lafont. The linear abstract machine. *Theoretical Computer Science*, 59:157–180, 1988. Some corrections in volume 62 (1988), pp. 327–328.

[22] Patrick Lincoln and John Mitchell. Operational aspects of linear lambda calculus. In *7th Symposium on Logic in Computer Science, IEEE*, pages 235–246. IEEE Computer Society Press, 1992.

[23] John C. Reynolds. Towards a theory of type structure. In *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer-Verlag, Paris, France, April 1974.

[24] John C. Reynolds. Types, abstraction, and parametric polymorphism. In R.E.A Mason, editor, *Information Processing*, pages 513–523. Elsevier Science Publishers B.V., 1983.

[25] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In *Proceedings of PARLE'94*, pages 398–413. Springer-Verlag, 1994. Lecture Notes in Computer Science number 817.

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

[27] Vasco T. Vasconcelos, Simon J. Gay, and António Ravara. Type checking a multithreaded functional language with session types. *Theoretical Computer Science*, 368(1–2):64–87, 2006.

[28] Edsko Vries, Rinus Plasmeijer, and David M. Abrahamson. Uniqueness typing simplified. In *Implementation and Application of Functional Languages: 19th International Workshop, IFL 2007, Freiburg, Germany, September 27-29, 2007. Revised Selected Papers*, pages 201–218, Berlin, Heidelberg, 2008. Springer-Verlag.

[29] Philip Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *Progarmming Concepts and Methods*, Sea of Galilee, Israel, April 1990. North Holland. IFIP TC 2 Working Conference.

[30] Philip Wadler. There's no substitute for linear logic. In *8th International Workshop on the Mathematical Foundations of Programming Semantics*, 1992.

[31] Philip Wadler. A taste of linear logic. In *Mathematical Foundations of Computer Science*, volume 711 of *Lecture Notes in Computer Science*, pages 185–210. Springer-Verlag, 1993.

[32] David Walker. A type system for expressive security policies. In *Proc. 27th ACM Symp. on Principles of Programming Languages (POPL)*, pages 254–267. ACM Press, Jan 2000.

[33] David Walker. *Advanced Topics in Types and Programming Languages*, chapter Substructural Type Systems. MIT Press, 2005.

[34] Keith Wansbrough and Simon Peyton Jones. Once upon a polymorphic type. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 15–28, New York, NY, USA, 1999. ACM.

[35] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994. Preliminary version in Rice TR 91-160.

[36] Dengping Zhu and Hongwei Xi. Safe Programming with Pointers through Stateful Views. In *Proceedings of the 7th International Symposium on Practical Aspects of Declarative Languages*, pages 83–97, Long Beach, CA, January 2005. Springer-Verlag LNCS vol. 3350.