1-1-2013

Linear Types, Protocols, and Processes in Classical FÂ°

Karl Mazurak
University of Pennsylvania, mazurak@cis.upenn.edu

Linear Types, Protocols, and Processes in Classical FÂ°

Abstract
Session types and typestate both promise a type system that can reason about protocol adherence. The complexity budgets of most programming languages, however, do not allow for new forms of types aimed at specific problem domains--even domains as broad as these.

Classical F◦ --read "F-pop"--is a typed λ-calculus based on classical (i.e., full) linear logic, wherein session types arise naturally from the interaction between the usual sums, products, and implications of linear logic and a simple process model, with the dualizing negation of classical logic naturally accounting for how a protocol is seen by each of a channel's endpoints. Classical F◦ expressions evaluate to processes, reminiscent of those in the π-calculus, that communicate over channels, but source expressions, rather than including processes and channels, employ only two novel control operators that account for process creation and communication.

F◦ is introduced by way of its intuitionistic fragment, which even on its own can account for typestate: the combination of linearity and polymorphism leads to natural encodings of many programmer-specified protocols. In fact, any protocol expressible as a regular language can be encoded in an intuitionistic F◦ type. F◦ distinguishes between linear and unrestricted types by using kinds together with a notion of subkinding, avoiding the pitfalls of approaches based on type qualifiers or modalities; kinds are related by a subkinding order that allows unrestricted types to be treated as though they were linear.

Soundness for intuitionistic and classical F◦ is proved both in the standard operational sense of preservation and progress and for an augmented semantics that shows more directly that the expected properties of linearity are preserved. This establishes the absence of deadlocks in closed, well-typed F◦ programs; it also guarantees that such programs will not "leak" processes as long as their result types are unrestricted.

LINEAR TYPES, PROTOCOLS, AND PROCESSES IN CLASSICAL F°

Karl Mazurak

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2013

Steve Zdancewic, Associate Professor of Computer and Information Science
Supervisor of Dissertation

Val Tannen, Professor of Computer and Information Science
Graduate Group Chairperson

Dissertation Committee

David Walker, Associate Professor of Computer Science

Jean Gallier, Professor of Computer and Information Science

Benjamin Pierce, Professor of Computer and Information Science

Stephanie Weirich, Associate Professor of Computer and Information Science

LINEAR TYPES, PROTOCOLS, AND PROCESSES IN CLASSICAL F°

Acknowledgements

Many people deserve thanks for helping me reach this stage, and I doubt my ability to properly thank them all. I would first like to express my gratitude to my committee—Stephanie Weirich, Benjamin Pierce, Jean Gallier, and David Walker—and especially to my advisor, Steve Zdancewic, for their guidance and patience. I could truly not ask for a better advisor than Steve, and I would not have made it even to the beginning of the work that led to this dissertation without him.

The University of Pennsylvania PL Club has been an amazing setting for the difficult journey of graduate school, and I would like to thank everyone involved for having made this a group that cares deeply about rigorous mathematics and yet remains warm and welcoming, a group with board game nights that feature only infrequent discussions of category theory. I would especially like to thank Jeff Vaughan, Jianzhou Zhao, Limin Jia, Luke Zarko, and Joseph Schorr, my coauthors over the years—along with Steve, they have firsthand experience with the patterns of my work, for good or for ill.

My parents, Peter and Elizabeth, deserve credit for first introducing me to computers, to the fact that computers can be programmed, and to the fact that this can be fun, and they supported my interests at a time when I could not have pursued those interests without assistance. I will not say that I've never felt unprepared in pursuing computer science, but, thanks to their early support, I have felt far less unprepared than I might have. I also owe thanks to my family as a whole for providing an environment where intellectual curiosity was always treated as worthwhile.

I first encountered real computer science thanks to Thomas Naps, whose teaching style managed to convey the significance of elegance in programming without shaming any of us for not already understanding this. Charles Fischer introduced me to typed functional

programming at the same time that Remzi Arpaci-Dusseau gave me the opportunity to learn about what can go wrong when programming in C; I doubt either of them intended for me to become a type theorist, but what's done is done. Tim Chevalier helped me research departments that might satisfy my new-found curiosity—and later helped me find the internship that inspired this dissertation—while Sheree Schrager's perseverance in the face of graduate school inadvertently inspired my own decision to pursue a PhD. After a difficult first year of graduate school, the Oregon Programming Languages Summer School provided the ideal setting to renew my excitement for type systems—as well as a chance to vent with Erika Rice Scherpelz.

I survived graduate school thanks to many friends in and around Philadelphia; I would like to thank—in addition to those whose names have already been mentioned—Diane Amoroso-O'Connor, Karena Arroyo, Sarah Cousins, Kuzman Ganchev, Andrew Hilton, Christina Jenkins, Kathryn Kennerly, Danielle Lancellotti, Andrea Piernock, Bryan and Michelle Segroves-Graves, Catherine Silverberg, Justin Smith, Nicholas Taylor, and others too numerous to list. I also owe thanks to my friends in Madison, especially Brian Aydemir, Debbie Chasman, Jessica Eanes, Joanna Lee, Mindy Preston, Cislyn Smith, Jessica Sorrell, and Zach Welhouse, all of whom have, at various times, gone along with a rather dubious claim of mine: that spending time with me while I write and edit this dissertation might somehow count as "fun".

My work at the University of Pennsylvania was supported by National Science Foundation grants CNS-0346939, CNS-0524059, and CCF-541040, but in order to find these numbers I had to look through my old publications, as at no point during my studies did I need to memorize any of these or many, many other logistical details. For this I must thank a great many employees of the Computer and Information Science department and of the School of Engineering and Applied Science—but in particular Michael Felker, whose valiant bureaucratic navigation leaves so many graduate students free to worry about their research rather than their paperwork.

ABSTRACT

LINEAR TYPES, PROTOCOLS, AND PROCESSES IN CLASSICAL F°

Karl Mazurak

Steve Zdancewic

Session types and typestate both promise a type system that can reason about protocol adherence. The complexity budgets of most programming languages, however, do not allow for new forms of types aimed at specific problem domains—even domains as broad as these.

Classical F°—read "F-pop"—is a typed λ-calculus based on classical (*i.e.*, full) linear logic, wherein session types arise naturally from the interaction between the usual sums, products, and implications of linear logic and a simple process model, with the dualizing negation of classical logic naturally accounting for how a protocol is seen by each of a channel's endpoints. Classical F° expressions evaluate to processes, reminiscent of those in the π-calculus, that communicate over channels, but source expressions, rather than including processes and channels, employ only two novel control operators that account for process creation and communication.

F° is introduced by way of its intuitionistic fragment, which even on its own can account for typestate: the combination of linearity and polymorphism leads to natural encodings of many programmer-specified protocols. In fact, any protocol expressible as a regular language can be encoded in an intuitionistic F° type. F° distinguishes between linear and unrestricted types by using kinds together with a notion of subkinding, avoiding the pitfalls of approaches based on type qualifiers or modalities; kinds are related by a subkinding order that allows unrestricted types to be treated as though they were linear.

Soundness for intuitionistic and classical F° is proved both in the standard operational sense of preservation and progress and for an augmented semantics that shows more directly that the expected properties of linearity are preserved. This establishes the absence of deadlocks in closed, well-typed F° programs; it also guarantees that such programs will not "leak" processes as long as their result types are unrestricted.

Contents

List of Figures

Chapter 1

Introduction

A programming language can be said to have a "complexity budget" limiting the distinct features that may productively be added to it: add too much and the language loses in everything from maintainability to accessibility. Simple features with diverse potential applications are thus highly desirable, and the core calculi present in the theoretical programming languages literature are more than vehicles for soundness proofs: they demonstrate the power of language features in isolation, making the case for their inclusion—following refinement by both theoreticians and pragmatists, of course—in progressively larger languages. But where are the budget-conscious to look for their good deals in complexity?

Given the origins of the typed λ-calculus, it is unsurprising that related calculi still draw much inspiration from the Curry-Howard isomorphism (Curry et al., 1958; Howard, 1980), the well-known correspondence between types in a programming language and propositions in a logic, and between programs—without general recursion or equivalent functionality—written in that language and proofs of the corresponding logical propositions. Time and time again the Curry-Howard isomorphism has led to elegant and useful programming language design—for example, System F (Girard, 1972; Reynolds, 1974) serves as both a proof language for a logic with second-order (but not first-order) quantification and as the canonical language with fully general parametric polymorphism.

Meanwhile, *linear logic* (Girard, 1987; Girard et al., 1989) restricts the properties of contraction (the ability to duplicate assumptions) and weakening (the ability to discard assumptions), and thus the propositions of linear logic can model physical resources in

straightforward ways. This immediately suggests utility in a programming language, and indeed ideas from linear logic were quick to appear in the programming languages literature, adopted first to eliminate garbage collection (Lafont, 1988) and shortly thereafter to handle mutable state (Wadler, 1990). Yet this interest has not translated into the implementation of linear types in even those languages favored by typed functional programmers.[1]

The case for linearity seems not to have been made well enough from the standpoint of the complexity budget. While many have demonstrated the power of linear types—be it through capability systems (Charguéraud and Pottier, 2008; Crary et al., 1999), explicit memory management and control of aliasing (Ahmed et al., 2007; Fähndrich and DeLine, 2002; Hicks et al., 2004; Vries et al., 2008; Zhu and Xi, 2005), or program analysis of changing state (DeLine and Fähndrich, 2001; Walker, 2000)—such work has not presented linearity as a simple language addition addressing a variety of needs. And, indeed, linear types can lead to awkward programming models and complicated language designs that are difficult both to implement and to use: a lack of implicit contraction and weakening by necessity makes many programs cumbersome to write, and the best means by which expressions of both linear and unrestricted type may coexist is not immediately obvious. Further, those formulations of linear logic that do not restrict themselves to its intuitionistic fragment are most naturally seen as sequent calculi, and when thus viewed as type systems they suggest term languages that differ significantly from standard λ-calculi.

This dissertation proposes F$^\circ$—pronounced "F-pop"—a linearly typed core calculus based on System F that aims to remedy the undersold status of linear typing. Rather than restrict itself to intuitionistic linear logic, F$^\circ$ demonstrates that the power of classical linear logic[2] is indeed compatible with λ-calculi. Linearity is particularly well suited, lacking unconstrained weakening and contraction, to enforcing *protocols* concerning how a term may be used, and the diverse applicability of F$^\circ$ will be demonstrated through the varied sorts of protocols it can enforce.

Chapter 2 introduces F$^\circ$ by way of its intuitionistic fragment, demonstrating how *kinds*

[1] Clean is often seen as the exception to this, but there are subtle differences between its uniqueness types, which consider how a value has been used (*i.e.*, whether it has been aliased), and standard linear types, which consider how a value may be used in the future.

[2] This use of "classical" in contrast to intuitionistic is common in the programming languages literature, though Girard uses "classical" in contrast to *linear* and prefers "full linear logic" where this dissertation uses "classical linear logic".

allow linear (kind $\circ$) and unrestricted (kind $\star$) types to mingle without complication thanks to a *subkinding* relation in which $\star \leqslant \circ$ (reflecting the idea that it is always safe to use something exactly once, even when it is not required). Intuitionistic F$^\circ$ lets programmers enforce their own protocol abstractions through the power of linearity and polymorphism—it allows for the straightforward encoding of *typestate* (Fähndrich and DeLine, 2002; Fähndrich et al., 2006), a desirable feature whenever, for instance, API calls must be made in a certain order—yet its typing discipline is lightweight enough to expose in a surface language: apart from the addition of kinds and the standard products and sums of intuitionistic linear logic, it is exactly System F.

Chapter 3 presents the full classical F$^\circ$, extending intuitionistic F$^\circ$ with the control operators **go** (spawning of a process) and **yield** (synchronizing on a channel), the type $\bot$ (process termination), and the kind $\bullet$ (strictly linear); at run time, classical F$^\circ$ expressions evaluate into processes in a very simple process calculus, and the standard linear connectives naturally double as *session types* (Honda et al., 1998; Takeuchi et al., 1994; Vasconcelos et al., 2006), with the duality of classical negation providing each channel endpoint with its appropriate communication protocol. Soundness of classical F$^\circ$ implies the absence of deadlocks in well-typed classical F$^\circ$ programs. The design of classical F$^\circ$ is reminiscent of the connection between **callcc** and classical logic (Griffin, 1990; Ong and Stewart, 1997) and specifically excludes the baffling (outside of a sequent calculus) $\parr$ connective of classical linear logic. Classical F$^\circ$ suggests linearity's applicability to systems in which frequent communication among logical processes—which may or may not correspond to physical processes or threads—is the norm (Armstrong et al., 1996; Haller and Odersky, 2009; Hickey, 2007), systems that are becoming more common as the need for concurrency grows.

Related work is discussed in greater detail at the end of Chapter 2 and Chapter 3, depending on the nature of the work in question. Chapter 4 discusses possible extensions to F$^\circ$ and their ramifications, and Chapter 5 presents a brief conclusion.

A purely multiplicative variant of intuitionistic F$^\circ$, there called System F$^\circ$, was presented at TLDI (Mazurak et al., 2010); the monomorphic, all-linear fragment of classical F$^\circ$ was presented at ICFP (Mazurak and Zdancewic, 2010) as Lolliproc.

Chapter 2

Intuitionistic F$^\circ$

While some of the most interesting applications of F$^\circ$ require the full power of classical linear logic, much is still possible within the intuitionistic fragment of the language—in particular, intuitionistic F$^\circ$'s type system can enforce user-defined protocols through the combination of polymorphism and linearity. Intuitionistic F$^\circ$ is also a very familiar setting in which to introduce F$^\circ$'s approach to linearity.

2.1 Terms and types

The syntax of intuitionistic F$^\circ$ is given in Figure 2.1; expressions e are categorized by types τ, which are in turn categorized by kinds κ. Intuitionistic F$^\circ$ has only two kinds: $\star$, pronounced "unrestricted", and $\circ$, pronounced "linear". The types of intuitionistic F$^\circ$, and their introduction and elimination forms, are mostly standard, but some subtleties will reveal their significance later on.

The function type in F$^\circ$ is written $\tau_1 \xrightarrow{\kappa} \tau_2$, where the kind annotation κ indicates how the function itself may be used: functions of type $\tau_1 \xrightarrow{\star} \tau_2$ may be used any number of times—that is, they may be duplicated or discarded—while those of type $\tau_1 \xrightarrow{\circ} \tau_2$ must be used exactly once.[1] To accommodate this kind annotation, the introduction form for functions is written $\lambda^\kappa x{:}\tau.\ e$; functions are eliminated by standard application $e_1\ e_2$.

[1] In the literature, $\multimap$ is the traditional symbol for the linear function type, but, when linear and unrestricted functions are present in the same system, there is disagreement over whether the use of $\multimap$ rather than $\to$ restricts the usage of the function value itself or its argument. F$^\circ$'s syntax avoids this confusion and will simplify matters in Chapter 3, when a third kind is introduced.

$$
\begin{array}{rcll}
\kappa & ::= & \star \mid \circ & \textit{kinds} \\
\tau & ::= & \alpha \mid \tau \xrightarrow{\kappa} \tau \mid \tau \,\&\, \tau \mid \forall \alpha{:}\kappa.\, \tau & \\
& \mid & \mathbf{1} \mid \tau \otimes \tau \mid \tau \oplus \tau \mid \exists \alpha{:}\kappa.\, \tau & \textit{types} \\
\\
i & ::= & 1 \mid 2 & \textit{indices} \\
e & ::= & x \mid \lambda^{\kappa} x{:}\tau.\, e \mid e\ e \mid \langle e, e \rangle \mid e.i \mid \Lambda \alpha{:}\kappa.\, v \mid e\ [\tau] & \textit{expressions} \\
& \mid & () \mid e; e \mid (e, e) \mid \mathbf{let}\ (x, x) = e\ \mathbf{in}\ e & \\
& \mid & \mathbf{inj}_i^{\tau \oplus \tau}\ e \mid \mathbf{case}\ e\ \mathbf{of}\ \mathbf{inj}_1\ x \mapsto e \mid \mathbf{inj}_2\ x \mapsto e & \\
& \mid & \mathbf{pack}\ \alpha{:}\kappa = \tau\ \mathbf{in}\ (e : \tau) \mid \mathbf{unpack}\ (\alpha, x) = e\ \mathbf{in}\ e & \\
\\
v & ::= & \lambda^{\kappa} x{:}\tau.\, e \mid \langle e, e \rangle \mid \Lambda \alpha{:}\kappa.\, v & \textit{values} \\
& \mid & () \mid (v, v) \mid \mathbf{inj}_i^{\tau \oplus \tau}\ v \mid \mathbf{pack}\ \alpha{:}\kappa = \tau\ \mathbf{in}\ (v : \tau) & \\
\\
E & ::= & [] \mid E\ e \mid v\ E \mid E\ [\tau] \mid E.i & \textit{evaluation contexts} \\
& \mid & E; e \mid (E, e) \mid (v, E) \mid \mathbf{let}\ (x, x) = E\ \mathbf{in}\ e & \\
& \mid & \mathbf{inj}_i^{\tau \oplus \tau}\ E \mid \mathbf{case}\ E\ \mathbf{of}\ \mathbf{inj}_1\ x \mapsto e \mid \mathbf{inj}_2\ x \mapsto e & \\
& \mid & \mathbf{pack}\ \alpha{:}\kappa = \tau\ \mathbf{in}\ (E : \tau) \mid \mathbf{unpack}\ (\alpha, x) = E\ \mathbf{in}\ e & \\
\\
\Gamma & ::= & \cdot \mid \Gamma, \alpha{:}\kappa \mid \Gamma, x{:}\tau & \textit{unrestricted typing contexts} \\
\Delta & ::= & \cdot \mid \Delta, x{:}\tau & \textit{linear typing contexts}
\end{array}
$$

Figure 2.1: Intuitionistic F°

The type $\tau_1 \,\&\, \tau_2$ is the *additive* product of τ_1 and τ_2—a curious choice of terminology (given that "additive" in other contexts generally refers to sums rather than products) that is nevertheless standard in the linear logic literature. The additive pair $\langle e_1, e_2 \rangle$ suspends e_1 and e_2; such pairs are eliminated with projection $e.1$ or $e.2$, after which evaluation of selected component continues. To preserve linearity, each additive pair must be eliminated exactly once, meaning that, in $\langle e_1, e_2 \rangle$, both e_1 and e_2 will depend on the same set of linear resources.

The universal types $\forall \alpha{:}\kappa.\, \tau$ gives a kind annotation κ to its type variable α; this annotation also appears in the type abstraction form $\Lambda \alpha{:}\kappa.\, v$. Type application is written as $e\ [\tau]$. Note that F° makes use of the *value restriction* (Wright and Felleisen, 1994), requiring that the body of a type abstraction be a value v—*i.e.*, a term that will not, on its own, take fur-

$$[\text{E-App}]\ (\lambda^{\kappa} x{:}\tau.\ e)\ v \longrightarrow \{x \mapsto v\}e \qquad [\text{E-Select}]\ \langle e_1, e_2\rangle.i \longrightarrow e_i$$

$$[\text{E-TApp}]\ (\Lambda\alpha{:}\kappa.\ v)\ [\tau] \longrightarrow \{\alpha \mapsto \tau\}v \qquad [\text{E-Seq}]\ ();e \longrightarrow e$$

$$[\text{E-Let}]\ \textbf{let}\ (x_1, x_2) = (v_1, v_2)\ \textbf{in}\ e \longrightarrow \{x_1 \mapsto v_1, x_2 \mapsto v_2\}e$$

$$[\text{E-Case}]\ \textbf{case}\ \textbf{inj}_i^{\tau_1 \oplus \tau_2}\ v\ \textbf{of}\ \textbf{inj}_1\ x_1 \mapsto e_1 \mid \textbf{inj}_2\ x_2 \mapsto e_2 \longrightarrow \{x_i \mapsto v\}e_i$$

$$[\text{E-Unpack}]\ \textbf{unpack}\ (\alpha, x) = (\textbf{pack}\ \alpha{:}\kappa = \tau\ \textbf{in}\ (v : \tau'))\ \textbf{in}\ e \longrightarrow \{\alpha \mapsto \tau, x \mapsto v\}e$$

$$[\text{E-Ctxt}]\ \frac{e \longrightarrow e'}{E\big[e\big] \longrightarrow E\big[e'\big]}$$

$$[\text{E}^*\text{-Id}]\ e \longrightarrow^* e \qquad [\text{E}^*\text{-Trans}]\ \frac{e \longrightarrow e' \qquad e' \longrightarrow^* e''}{e \longrightarrow^* e''}$$

Figure 2.2: Evaluation rules for intuitionistic F°

ther evaluation steps. This is a common restriction in languages that mix polymorphism with other features—*e.g.*, ML—and it turns out to be critical to the soundness of F°.

The remaining constructs of intuitionistic F° are entirely straightforward. The unit type $\mathbf{1}$ has a single element $()$ and is eliminated by sequencing $e_1; e_2$. The *multiplicative* product $\tau_1 \otimes \tau_2$ is more familiar than its additive cousin: a multiplicative pair (e_1, e_2) eagerly evaluates its components, and they must depend on disjoint sets of linear resources; the elimination form $\textbf{let}\ x_1, x_2 = e\ \textbf{in}\ e'$ binds both components of such pairs, as projection in this case could violate linearity.

Sums $\tau_1 \oplus \tau_2$ are completely standard, introduced by injection $\textbf{inj}_i^{\tau_1 \oplus \tau_2}\ e$ and eliminated with case analysis $\textbf{case}\ e\ \textbf{of}\ \textbf{inj}_1\ x_1 \mapsto e_1 \mid \textbf{inj}_2\ x_2 \mapsto e_2$. F° uses superscript type annotations in cases like $\textbf{inj}_i^{\tau_1 \oplus \tau_2}\ e$ where additional type information is necessary for typechecking but serves no further purpose—in this case, the type of the argument reflects only one side of the sum type, so the annotation fills in the missing information.[2]

Finally, F° features *existential types* $\exists\alpha{:}\kappa.\ \tau$, which differ from their standard presen-

[2]One might instead introduce, in this case, polymorphic $\textbf{inj}_1$ and $\textbf{inj}_2$ terms and rely on type application to instantiate them at the appropriate types. Doing so, however, would clutter the language with partially applied constants, and there does not seem to be anything that F° might gain from this approach to justify said clutter.

$$[\text{S-REFL}]\ \kappa \leqslant \kappa \qquad [\text{S-TOP}]\ \kappa \leqslant \circ \qquad [\text{K-SUB}]\ \frac{\Gamma \vdash \tau : \kappa_1 \qquad \kappa_1 \leqslant \kappa_2}{\Gamma \vdash \tau : \kappa_2}$$

$$[\text{K-TVAR}]\ \frac{\alpha{:}\kappa \in \Gamma}{\Gamma \vdash \alpha : \kappa} \qquad [\text{K-ARROW}]\ \frac{\Gamma \vdash \tau_1 : \kappa_1 \qquad \Gamma \vdash \tau_2 : \kappa_2}{\Gamma \vdash \tau_1 \xrightarrow{\kappa} \tau_2 : \kappa}$$

$$[\text{K-WITH}]\ \frac{\Gamma \vdash \tau_1 : \kappa_1 \qquad \Gamma \vdash \tau_2 : \kappa_2}{\Gamma \vdash \tau_1 \,\&\, \tau_2 : \circ} \qquad [\text{K-ALL}]\ \frac{\Gamma, \alpha{:}\kappa \vdash \tau : \kappa' \qquad \alpha \notin \text{dom}(\Gamma)}{\Gamma \vdash \forall \alpha{:}\kappa.\, \tau : \kappa'}$$

$$[\text{K-UNIT}]\ \Gamma \vdash \mathbf{1} : \star \qquad [\text{K-TENSOR}]\ \frac{\Gamma \vdash \tau_1 : \kappa \qquad \Gamma \vdash \tau_2 : \kappa}{\Gamma \vdash \tau_1 \otimes \tau_2 : \kappa}$$

$$[\text{K-PLUS}]\ \frac{\Gamma \vdash \tau_1 : \kappa \qquad \Gamma \vdash \tau_2 : \kappa}{\Gamma \vdash \tau_1 \oplus \tau_2 : \kappa} \qquad [\text{K-EXISTS}]\ \frac{\Gamma, \alpha{:}\kappa \vdash \tau : \kappa' \qquad \alpha \notin \text{dom}(\Gamma)}{\Gamma \vdash \exists \alpha{:}\kappa.\, \tau : \kappa'}$$

Figure 2.3: Kinding rules for intuitionistic F°

tation only in the presence of the kind annotation κ. Existential types allow, essentially, for pairs in which the first element is a type and the second is a term; the term may depend on the structure of the type, but the enclosing context can only know that type as an opaque type variable. Existentials are introduced, packaging together a term and a type, with **pack** $\alpha{:}\kappa = \tau$ **in** $(e : \tau')$; the type annotation τ' indicates which occurrences of τ within the type of e should be abstracted to α. Existential packages are opened with **unpack** $(\alpha, x) = e$ **in** e'; within e', the type and term components of the existential package e are bound to α and x respectively.

Intuitionistic F°'s call-by-value operational semantics can be seen in Figure 2.2. It is entirely standard and kind-agnostic; each elimination form reacts with its associated introduction form. All other evaluation is accounted for, in rule E-CTXT, by decomposing an expression into an *evaluation context* E applied to a subexpression e: if e steps to e', then $E\left[e\right]$ steps to $E\left[e'\right]$. The definition of evaluation contexts is also completely standard and reflects the usual definition of call-by-value reduction—*e.g.*, the left side of an application is reduced before the right and values are exactly those expressions that cannot reduce further. At this point evaluation contexts merely serve as a convenience, removing the need for a dozen congruence rules; in Chapter 3, however, they will become indispensable.

To understand how types are assigned to intuitionistic F$^\circ$ expressions, we must first look at how kinds are assigned to types. As mentioned, the kind $\star$ classifies unrestricted types, the members of which may be duplicated or discarded, whereas $\circ$ classifies linear types, the members of which must be used exactly once. But note that the behavior allowed for expressions of linear type is included in the behavior allowed for those of unrestricted type; it should thus be sound to classify an unrestricted type as linear. In F$^\circ$, this idea is captured through a *subkinding* judgment, written $\kappa_1 \leqslant \kappa_2$, that yields a very simple subkinding order:

$$\leqslant \qquad \begin{array}{c} \circ \\ | \\ \star \end{array}$$

Here F$^\circ$ diverges from many linear systems that make memory management their first priority (Wadler, 1990; Ahmed et al., 2007; Fähndrich and DeLine, 2002; Hicks et al., 2004): if linearity is meant to imply that memory can be used in a way that might not be safe otherwise, then clearly unrestricted should *not* be a subkind of linear. (However, while F$^\circ$ is not built around memory management in that sense, we will see in Section 2.2.2 that its constructs are indeed well suited for ensuring that memory is used correctly.)

Figure 2.3 shows the kinding rules for intuitionistic F$^\circ$, where the kinding judgment is written $\Gamma \vdash \tau : \kappa$. Rule K-SUB incorporates subkinding as described above, while K-TVAR, as one would expect, looks up a type variable's kind in the context Γ. The next three rules, K-ARROW, K-WITH, and K-ALL, each deal with a construct that suspends computation, and each receives slightly different treatment. The kind of $\tau_1 \xrightarrow{\kappa} \tau_2$ is κ regardless of the kinds of τ_1 and τ_2. The kind of $\tau_1 \,\&\, \tau_2$, on the other hand, is always $\circ$—more permissive kinding in this case would actually complicate soundness, and the functionality of unrestricted additive pairs can be achieved with multiplicative pairs of unrestricted functions. Finally, the kind of $\forall\alpha{:}\kappa.\ \tau$ is simply the kind of τ under the appropriate context extension—it is in fact this design decision that gives F$^\circ$ the value restriction, as will become clear in the discussion of type soundness.

The remaining kinding rules are all straightforward. K-UNIT declares the unit type $\mathbf{1}$ to be unrestricted for reasons of convenience, as F$^\circ$ will remain sound in either case. The rules K-TENSOR, K-PLUS, and K-EXISTS all require that the kinds of any type components

$$[\text{B-LIN}]\ \frac{\Gamma \vdash \tau : \circ \qquad \Gamma; \Delta, x{:}\tau \vdash e{:}\tau' \qquad x \notin \Gamma, \Delta}{[\Gamma; \Delta], x{:}\tau \vdash e{:}\tau'}$$

$$[\text{B-UN}]\ \frac{\Gamma \vdash \tau : \star \qquad \Gamma, x{:}\tau; \Delta \vdash e{:}\tau' \qquad x \notin \Gamma, \Delta}{[\Gamma; \Delta], x{:}\tau \vdash e{:}\tau'}$$

$$[\text{U-EMPTY}]\ \cdot \Cup \cdot = \cdot$$

$$[\text{U-LEFT}]\ \frac{\Delta_1 \Cup \Delta_2 = \Delta \qquad x \notin \Delta}{\Delta_1, x{:}\tau \Cup \Delta_2 = \Delta, x{:}\tau} \qquad [\text{U-RIGHT}]\ \frac{\Delta_1 \Cup \Delta_2 = \Delta \qquad x \notin \Delta}{\Delta_1 \Cup \Delta_2, x{:}\tau = \Delta, x{:}\tau}$$

Figure 2.4: Auxiliary judgments for intuitionistic F° typing

match the kind of the type being constructed. Combined with subkinding, this gives the expected result in the cases of sums and multiplicative products: to be given the unrestricted kind, both sides must be unrestricted; if either the left or the right side can only be linear, then the entire construct must be linear. Note that, while the presence of subkinding makes kinding nondeterministic, it is clearly decidable and the simple subkinding lattice makes it easy to define a type's most specific kind.

While the kinding rules required just a single context Γ—with no special consideration needed concerning how often it appears in a rule's premises—the typing rules depend on both this unrestricted Γ, which contains type variables and unrestricted expression variables, and a linear context Δ, which contains linear expression variables. This two-context presentation follows Barber's DILL (Barber, 1997) and leads to a simpler metatheory as compared to a single-context presentation. Due to subkinding, a variable of an unrestricted type may be treated as though its type were linear and must be permitted to be bound in either Γ or Δ. This is expressed concisely with the notation $[\Gamma; \Delta], x{:}\tau \vdash e : \tau'$, indicating that either Γ or Δ is extended with the binding for x for purposes of typechecking e. Rule B-UN in Figure 2.4 requires that an extension of Γ bind x to an unrestricted τ, while subkinding allows any well-formed τ to be accepted by B-LIN. Additionally, the linear resources in Δ must be split whenever two subexpressions will both be evaluated—*e.g.*, in $e_1\ e_2$ and (e_1, e_2) but not in $\langle e_1, e_2 \rangle$. Such a linear context split into Δ_1 and Δ_2 is written $\Delta_1 \Cup \Delta_2$; this splitting relation is likewise defined in Figure 2.4.

$$[\text{T-LVAR}]\ \Gamma; x{:}\tau \vdash x : \tau \qquad\qquad [\text{T-UVAR}]\ \frac{x{:}\tau \in \Gamma}{\Gamma; \cdot \vdash x : \tau}$$

$$[\text{T-LAM}]\ \frac{[\Gamma; \Delta], x{:}\tau_1 \vdash e : \tau_2 \qquad \kappa = \star \implies \Delta = \cdot}{\Gamma; \Delta \vdash \lambda^{\kappa} x{:}\tau_1.\ e : \tau_1 \xrightarrow{\kappa} \tau_2}$$

$$[\text{T-APP}]\ \frac{\Gamma; \Delta_1 \vdash e_1 : \tau_1 \xrightarrow{\kappa} \tau_2 \qquad \Gamma; \Delta_2 \vdash e_2 : \tau_1}{\Gamma; \Delta_1 \Cup \Delta_2 \vdash e_1\ e_2 : \tau_2}$$

$$[\text{T-APAIR}]\ \frac{\Gamma; \Delta \vdash e_1 : \tau_1 \qquad \Gamma; \Delta \vdash e_2 : \tau_2}{\Gamma; \Delta \vdash \langle e_1, e_2 \rangle : \tau_1 \,\&\, \tau_2} \qquad [\text{T-SELECT}]\ \frac{\Gamma; \Delta \vdash e : \tau_1 \,\&\, \tau_2}{\Gamma; \Delta \vdash e.i : \tau_i}$$

$$[\text{T-TLAM}]\ \frac{\Gamma, \alpha{:}\kappa; \Delta \vdash v : \tau \qquad \alpha \notin \Gamma}{\Gamma; \Delta \vdash \Lambda\alpha{:}\kappa.\ v : \forall\alpha{:}\kappa.\ \tau} \qquad [\text{T-TAPP}]\ \frac{\Gamma; \Delta \vdash e : \forall\alpha{:}\kappa.\ \tau' \qquad \Gamma \vdash \tau : \kappa}{\Gamma; \Delta \vdash e\ [\tau] : \{\alpha \mapsto \tau\}\tau'}$$

$$[\text{T-UNIT}]\ \Gamma; \cdot \vdash () : \mathbf{1} \qquad\qquad [\text{T-SEQ}]\ \frac{\Gamma; \Delta_1 \vdash e_1 : \mathbf{1} \qquad \Gamma; \Delta_2 \vdash e_2 : \tau}{\Gamma; \Delta_1 \Cup \Delta_2 \vdash e_1; e_2 : \tau}$$

$$[\text{T-MPAIR}]\ \frac{\Gamma; \Delta_1 \vdash e_1 : \tau_1 \qquad \Gamma; \Delta_2 \vdash e_2 : \tau_2}{\Gamma; \Delta_1 \Cup \Delta_2 \vdash (e_1, e_2) : \tau_1 \otimes \tau_2}$$

$$[\text{T-LET}]\ \frac{\Gamma; \Delta_1 \vdash e_1 : \tau_1 \otimes \tau_2 \qquad [\Gamma; \Delta_2], x_1{:}\tau_1, x_2{:}\tau_2 \vdash e_2 : \tau}{\Gamma; \Delta_1 \Cup \Delta_2 \vdash \mathbf{let}\ (x_1, x_2) = e_1\ \mathbf{in}\ e_2 : \tau}$$

$$[\text{T-IN}]\ \frac{\Gamma; \Delta \vdash e : \tau_i \qquad \Gamma \vdash \tau_1 : \kappa_1 \qquad \Gamma \vdash \tau_2 : \kappa_2}{\Gamma; \Delta \vdash \mathbf{inj}_i^{\tau_1 \oplus \tau_2}\ e : \tau_1 \oplus \tau_2}$$

$$[\text{T-CASE}]\ \frac{\Gamma; \Delta_1 \vdash e : \tau_1 \oplus \tau_2 \qquad [\Gamma; \Delta_2], x_1{:}\tau_1 \vdash e_1{:}\tau \qquad [\Gamma; \Delta_2], x_2{:}\tau_2 \vdash e_2{:}\tau}{\Gamma; \Delta_1 \Cup \Delta_2 \vdash \mathbf{case}\ e\ \mathbf{of}\ \mathbf{inj}_1\ x_1 \mapsto e_2 \mid \mathbf{inj}_2\ x_2 \mapsto e_2 : \tau}$$

$$[\text{T-PACK}]\ \frac{\Gamma; \Delta \vdash e : \{\alpha \mapsto \tau\}\tau' \qquad \Gamma \vdash \tau : \kappa}{\Gamma; \Delta \vdash \mathbf{pack}\ \alpha{:}\kappa = \tau\ \mathbf{in}\ (e : \tau') : \exists\alpha{:}\kappa.\ \tau'}$$

$$[\text{T-UNPACK}]\ \frac{\Gamma; \Delta_1 \vdash e_1 : \exists\alpha{:}\kappa.\ \tau \qquad [\Gamma, \alpha{:}\kappa; \Delta_2], x{:}\tau \vdash e_2 : \tau'}{\Gamma; \Delta_1 \Cup \Delta_2 \vdash \mathbf{unpack}\ (\alpha, x) = e_1\ \mathbf{in}\ e_2 : \tau'}$$

Figure 2.5: Typing rules for intuitionistic F°

Intuitionistic F°'s typing judgment, written $\Gamma; \Delta \vdash e : \tau$, is defined in Figure 2.5. The variable rules T-LVAR and T-UVAR handle variables from the linear and unrestricted contexts respectively. Note that, while both rules allow an arbitrary Γ—permitting weakening of unrestricted assumptions—Δ may contain no variables aside from, in the linear case, the variable whose type is being checked. T-LAM demonstrates the non-deterministic extension $[\Gamma; \Delta], x{:}\tau$ in typechecking the body of a function; it further requires, if the entire expression is meant to be unrestricted, that the body not depend on any linear variables. This last restriction is essential in making sure that it is safe to duplicate or discard a function value of an unrestricted type.

The remaining typing rules are all fairly straightforward, but there are a few more details of interest. First, the only circumstances under which a linear context is duplicated are those in which only one subexpression typechecked with the duplicated context will be evaluated at run time—*e.g.*, in T-APAIR. The more common case is for a linear context to be split into $\Delta_1 \Cup \Delta_2$, as in T-APP, T-SEQ, T-MPAIR, T-LET, and T-UNPACK. T-CASE does both, using Δ_1 to typecheck the expression being analyzed and Δ_2 to typecheck the branches, only one of which will ultimately be evaluated. Finally, both T-PACK and T-TAPP make implicit use of subkinding: polymorphic expressions expecting a linear type may be given an unrestricted one, and unrestricted types may be hidden in an existential package exposing a linear type variable.

Weakening (neglecting to use a variable) and contraction (using a variable more than once) are only possible with variables of unrestricted type. Rather than add explicit contraction and weakening rules, these properties are built 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 every rule with multiple premises duplicates Γ but handles Δ as described above. Weakening and contraction thus follow naturally:

Property 1 (Weakening). *If* $\Gamma_1, \Gamma_2; \Delta \vdash e : \tau'$, $\Gamma_1 \vdash \tau : \star$, *and* x *is not in* Γ_1, Γ_2, *or* Δ, *then* $\Gamma_1, x{:}\tau, \Gamma_2; \Delta \vdash e : \tau'$.

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

From these rules we can see that the type $\forall\alpha{:}\circ.\ \forall\beta{:}\star.\ \alpha \xrightarrow{\star} \beta \xrightarrow{\star} \alpha$, for instance, is uninhabited—there is no way to write the requisite unrestricted inner function in the presence of a linear variable of type α. This is as it should be, because even though it ostensibly promises to return its argument of type α, the partial application of a function of this type could be freely replicated or discarded, violating linearity. The same is true for the type $\forall\alpha{:}\circ.\ \forall\beta{:}\star.\ \alpha \xrightarrow{\circ} \beta \xrightarrow{\star} \alpha$, but the types $\forall\alpha{:}\circ.\ \forall\beta{:}\star.\ \alpha \xrightarrow{\star} \beta \xrightarrow{\circ} \alpha$ and $\forall\alpha{:}\circ.\ \forall\beta{:}\star.\ \beta \xrightarrow{\star} \alpha \xrightarrow{\star} \alpha$ are both inhabited, as their inhabitants behave appropriately with respect to linearity.

It is easy to see that, modulo the value restriction, intuitionistic F$^\circ$ is an extension of System F. In fact, an earlier presentation (Mazurak et al., 2010) of a core of intuitionistic F$^\circ$, there called System F$^\circ$, extended System F with only kind annotations, leveraging well-known encodings for existentials and multiplicative products while omitting the additive constructs. This is not the approach taken here, for reasons that will become clear in Chapter 3, but the encodings of System F remain useful. For instance, general **let** expressions can be defined as

$$\textbf{let}\ x = e\ \textbf{in}\ e' \quad \triangleq \quad (\lambda^\circ x{:}\tau.\ e')\ e$$

where τ is the type of e—whenever **let** expressions are used in examples, τ should be obvious from context. Note that, since the function representing the body of the let is always used immediately and exactly once, it is a natural candidate for the linear function type, allowing **let** expressions to freely typecheck in the presence of linear free variables.

The standard System F encoding of natural number is also available, providing unrestricted values and making use of unrestricted functions while still allowing an arbitrary (potentially linear) type as the result of the elimination of a natural number:

$$\mathsf{Nat} \quad \triangleq \quad \forall\alpha{:}\circ.\ (\alpha \xrightarrow{\star} \alpha) \xrightarrow{\star} \alpha \xrightarrow{\star} \alpha$$

As usual, $n\ [\tau]\ f\ x$ applies the function f (which takes τ to τ) to x a total of n times:

$$\begin{aligned}
0 &\triangleq \Lambda\alpha{:}\circ.\ \lambda^\star s{:}\alpha \xrightarrow{\star} \alpha.\ \lambda^\star z{:}\alpha.\ z \\
1 &\triangleq \Lambda\alpha{:}\circ.\ \lambda^\star s{:}\alpha \xrightarrow{\star} \alpha.\ \lambda^\star z{:}\alpha.\ s\ z \\
2 &\triangleq \Lambda\alpha{:}\circ.\ \lambda^\star s{:}\alpha \xrightarrow{\star} \alpha.\ \lambda^\star z{:}\alpha.\ s\ (s\ z) \\
&\vdots
\end{aligned}$$

It is instructive to look at the kind annotations in this definition and consider the functions they serve. As α is the type argument for eliminating a natural number, giving α the kind $\circ$ simply makes natural numbers as general as possible. The $\star$ annotation on each λ ensure that natural numbers themselves are unrestricted—were the second annotation $\circ$, unrestricted natural numbers would become linear when partially applied to a successor-eliminating function, and changing both to $\circ$ would give us natural numbers that must be treated linearly.

Replace the $\star$ on the inner arrow—i.e., the type of s—with $\circ$, however, and (in addition to requiring a $\circ$ on the second arrow and ruling out partial application) we no longer have natural numbers at all, as only the term representing 1 remains typeable: a linear argument like s must be applied exactly once, so given z of type τ and s of type $\tau \xrightarrow{\circ} \tau$, the only well-typed expressions of type τ are equivalent to $s\ z$. While this type is not particularly useful, it suggests other possibilities. Consider, for example

$$\forall\alpha{:}\circ.\ (\alpha \xrightarrow{\circ} \alpha) \xrightarrow{\star} (\alpha \xrightarrow{\circ} \alpha) \xrightarrow{\circ} (\alpha \xrightarrow{\circ} \alpha) \xrightarrow{\circ} \alpha \xrightarrow{\circ} \alpha$$

Its inhabitants represent the different possible orders in which the three function arguments can be applied to an argument, giving this type six canonical members. More complex types in this vein, incorporating multiple type variables and both linear and unrestricted functions, can have an unbounded number of inhabitants while still enforcing, thanks to linearity and polymorphism, restrictions on how often or in which order certain functions are applied. The next section demonstrates this with some more realistic examples.

2.2 Examples

We will now see the applicability of intuitionistic F$^\circ$ through three sets of examples. First, an encoding of a safe filesystem establishes the basic techniques used to extract good behavior from a mix of linearity and parametric polymorphism. Second, a variety of reference cell implementations show that, while F$^\circ$ does not build in notions of references and aliasing, protocols defining correct memory management can indeed be enforced by its types. Third, it is proved that *any* protocol expressible as a finite automaton has a

corresponding F° type—in addition to establishing that a large class of protocols can be encoded in our type system, this proof highlights intuitions about linearity that still need formalization, which are addressed in Section 2.3.2.

These examples all include records, always containing only fields of unrestricted type, eliminated by projection—these records are often analogous to dictionaries in implementations of Haskell typeclasses (Wadler and Blott, 1989). Such records add significant readability and can clearly be encoded in terms of the multiplicative pairs. The single variable **let** defined in the previous section also sees frequent use, and n-ary existential types are used for simplicity's sake.

2.2.1 A linear filesystem

Linearity lets us specify a filesystem interface that requires each filehandle be closed exactly once and forbids its use thereafter. Given the base types String and Char—but without any understanding of linearity—an operating system might provide a type UnsafeFH and its associated file operations with the following augmented F° signatures[3]:

$$\begin{aligned}
\mathsf{UnsafeFH} &: \star \\
\\
\mathsf{unsafe_open} &: \mathsf{String} \xrightarrow{\star} \mathsf{UnsafeFH} \\
\mathsf{unsafe_read} &: \mathsf{UnsafeFH} \xrightarrow{\star} \mathsf{Char} \\
\mathsf{unsafe_write} &: \mathsf{Char} \xrightarrow{\star} \mathsf{UnsafeFH} \xrightarrow{\star} \mathbf{1} \\
\mathsf{unsafe_close} &: \mathsf{UnsafeFH} \xrightarrow{\star} \mathbf{1}
\end{aligned}$$

Here UnsafeFH is analogous to a Unix file descriptor or FILE pointer, and there are no static guarantees that a filehandle of this type is not read from or written to after it has been closed. Intuitionistic F°, however, makes it easy to create a safe interface protecting filehandles from misuse. First, taking α to be the abstract filehandle type, let us define a record type $\mathsf{FH}(\alpha)$ of safe file operations:

$$\begin{aligned}
\mathsf{FH}(\alpha) \triangleq \{\ & \mathsf{read} : \alpha \xrightarrow{\star} \mathsf{Char} \otimes \alpha \\
& \mathsf{write} : \mathsf{Char} \xrightarrow{\star} \alpha \xrightarrow{\star} \alpha, \\
& \mathsf{close} : \alpha \xrightarrow{\star} \mathbf{1}\ \}
\end{aligned}$$

[3]This example uses roughly the same interface given by DeLine and Fähndrich (2001) to motivate Vault and discussed by Kiselyov and Shan (2008) as an alternative to their filehandle regions.

Now open can be defined to return an existential package, hiding the filehandle's true type behind a linear type variable and pairing the filehandle with its associated operations:

$$
\begin{aligned}
&\mathsf{open} \;:\; \mathsf{String} \xrightarrow{\star} \exists \alpha{:}\circ.\ \alpha \otimes \mathsf{FH}(\alpha) \\
&\mathsf{open} \;\triangleq\; \lambda^{\star} f{:}\mathsf{String}.\ \textbf{let}\ \mathit{handle} = \mathsf{unsafe\_open}\, f\ \textbf{in} \\
&\qquad \textbf{pack}\ \alpha = \mathsf{UnsafeFH}\ \textbf{in} \\
&\qquad (\ \mathit{handle}, \\
&\qquad\quad \{\ \mathsf{read} = \lambda^{\star} h{:}\mathsf{UnsafeFH}.\ (\mathsf{unsafe\_read}\ h, h), \\
&\qquad\quad\ \ \mathsf{write} = \lambda^{\star} c{:}\mathsf{Char}.\ \lambda^{\star} h{:}\mathsf{UnsafeFH}. \\
&\qquad\quad\qquad\qquad \mathsf{unsafe\_write}\ c\ h; h, \\
&\qquad\quad\ \ \mathsf{close} = \mathsf{unsafe\_close} \\
&\qquad\quad \}\ ) : \alpha \otimes \mathsf{FH}(\alpha)
\end{aligned}
$$

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, which it does, then any use of the operations provided by an existential package it creates must treat them correctly as well.

After unpacking a filehandle, a program may read or write from it simply by invoking the provided member functions, but good programming practice requires the ability to define new functions over filehandles. The separation of the linear filehandle from the unrestricted file operations makes it easy to write functions like read_line, which can be given the following type:

$$\mathsf{read\_line} : \forall \alpha{:}\circ.\ \mathsf{FH}(\alpha) \xrightarrow{\star} \alpha \xrightarrow{\star} \mathsf{String} \otimes \alpha$$

This type suggests a usage pattern wherein many file operations occur within the scope of a single **unpack**; such a pattern allows for types that reflect, for instance, the fact that the filehandle returned by read_line is the same one that it was given.

Writing functions this way allows for useful type coercions; for example, suppose we are also able to open files in read-only mode, yielding restricted existential packages of the form

$$
\begin{aligned}
\mathsf{ROFH}(\alpha) \;=\; \{\ &\mathsf{read} : \alpha \xrightarrow{\star} \mathsf{Char} \otimes \alpha, \\
&\mathsf{close} : \alpha \xrightarrow{\star} \mathbf{1}\ \}
\end{aligned}
$$

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

Such a coercion from a record type with more members to a record type with fewer is implicit in systems with structural subtyping and mirrors some of the simpler relationships that can arise between Haskell typeclasses (Wadler and Blott, 1989). The use of existential quantification in tying methods to their owners is reminiscent of object encodings in languages with structural subtyping (Bruce et al., 1999).

2.2.2 Reference cells

An interface for unshareable linear reference cells looks remarkably similar to the filehandle interface just presented. A reference type is already naturally thought of as parameterized over the type of its contents, however; let us distinguish between such ordinary (and often unrestricted) type parameters from the linear *capability parameters* by writing, for instance, $\mathsf{UniqRef}[\tau](\alpha)$ for the type of unshareable reference cells containing values of type τ and making use of a capability—an opaque value—of the type α. Such a reference cell could be typed as follows:

$$\begin{aligned}\mathsf{UniqRef}[\tau](\alpha) \triangleq \{\ & \mathsf{set} : \tau \xrightarrow{\star} \alpha \xrightarrow{\star} \alpha, \\ & \mathsf{get} : \alpha \xrightarrow{\star} \tau \otimes \alpha, \\ & \mathsf{free} : \alpha \xrightarrow{\star} \mathbf{1}\ \}\end{aligned}$$

$$\mathsf{uniq\_ref} \ : \ \forall\beta{:}\star.\ \beta \xrightarrow{\star} \exists\alpha{:}\circ.\ \alpha \otimes \mathsf{Ref}[\beta](\alpha)$$

On its own, this is not particularly interesting, as such a linear reference cell simply encodes the practice of threading a value through a program—indeed, an implementation of uniq_ref could very well just equate α and τ. Instead, however, let us consider variations on UniqRef that make more sense as safe interfaces wrapping operations over a type UnsafeRef of manually managed memory, much as File in the previous wraps potentially unsafe filehandle operations.

While the above UniqRef could be such a safe interface, a more obvious one—which

itself requires no linearity—is the type GCRef:

$$\mathsf{GCRef}[\tau](\alpha) \triangleq \{\, \mathsf{set} : \tau \xrightarrow{\star} \alpha \xrightarrow{\star} \mathbf{1},\ \mathsf{get} : \alpha \xrightarrow{\star} \tau \,\}$$

$$\mathsf{gc\_ref} \ : \ \forall\beta{:}\star.\ \beta \xrightarrow{\star} \exists\alpha{:}\star.\ \alpha \otimes \mathsf{GCRef}[\beta](\alpha)$$

The operations included in a record of type $\mathsf{GCRef}[\tau](\alpha)$ allow the retrieval and assignment of the contents of the underlying reference cell of type τ but do not provide a means of freeing this memory, as is usually seen in languages with garbage collection. Even if manually managed reference cells are also present, hiding the reference type behind α—which may be unrestricted in this example—ensures that a garbage-collected reference cell cannot mistakenly be freed.

This, too, is not particularly exciting. Intuitionistic F$^\circ$, however, works equally well with references that begin their lives manually managed (and thus of linear type) but later are put under the garbage collector's control. $\mathsf{UniqRef}[\tau](\alpha)$ simply needs an additional function to serve as the appropriate coercion:

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

By consuming and not returning α, gc prevents free from being called on the now garbage-collected (but unrestricted) reference, even if, in actuality, both parameters are simply bound to the underlying reference type. We now have a coercion from alias-free to potentially aliased pointers, a fact that, were linearity conflated with alias-freedom, would run counter to the subkinding relation of $\star \leqslant \circ$.

Going even further, an intermediate point between a strictly linear and a garbage collected reference is a reference that must be explicitly aliased and whose aliases must be explicitly discarded—*i.e.*, the cell is managed using reference counting, but the operations that change the reference count are made explicit. This can also be defined easily in intu-

itionistic F°:

$$\mathsf{RCRef}[\tau](\alpha) \triangleq \{\ \mathsf{set} : \tau \xrightarrow{\star} \alpha \xrightarrow{\star} \alpha,\ \mathsf{get} : \alpha \xrightarrow{\star} \tau \otimes \alpha,\ \mathsf{alias} : \alpha \xrightarrow{\star} \alpha \otimes \alpha,\ \mathsf{drop} : \alpha \xrightarrow{\star} \mathbf{1}\ \}$$

$$\mathsf{rc\_ref} \ : \ \forall \beta{:}\star.\ \beta \xrightarrow{\star} \exists \alpha{:}\circ.\ \alpha \otimes \mathsf{RCRef}[\beta](\alpha)$$

A straightforward implementation of rc_ref could use as its capability a pair of the reference cell to be managed and a second, integer-valued cell to act as a counter to be adjusted by the alias and drop operations. Both the main cell and the counter cell could safely be freed when the count reaches zero.

However, while RCRef still provides a linear access capability, it is no longer possible to know for certain that there are no other outstanding references to the cell in question. To remedy this, as is often done in capability calculi (Ahmed et al., 2007; Charguéraud and Pottier, 2008), we can give our cells both an exclusive capability parameter α and a shared capability parameter β, with possession of the exclusive access capability implying that there currently exist no copies of the shared access capability. If, for example, the cell contents should not be altered if any aliases exist, this can be accomplished with

$$\begin{aligned}\mathsf{ShareRef}[\tau](\alpha,\beta) \triangleq \{\ & \mathsf{set\_excl} : \tau \xrightarrow{\star} \alpha \xrightarrow{\star} \alpha,\\ & \mathsf{get\_excl} : \alpha \xrightarrow{\star} \tau \otimes \alpha)\\ & \mathsf{get\_shared} : \beta \xrightarrow{\star} \tau \otimes \beta,\\ & \mathsf{share} : \alpha \xrightarrow{\star} \beta,\\ & \mathsf{claim} : \beta \xrightarrow{\star} \alpha \oplus \beta,\\ & \mathsf{alias} : \beta \xrightarrow{\star} \beta \otimes \beta,\\ & \mathsf{drop} : \beta \xrightarrow{\star} \mathbf{1},\\ & \mathsf{free} : \alpha \xrightarrow{\star} \mathbf{1}\ \}\end{aligned}$$

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

The return type of claim is $\alpha \oplus \beta$, meaning that the caller of claim, after relinquishing the shared capability, needs to check whether exclusive access was claimed successful; if it was

not, shared access remains. Exclusive access, of course, should only be granted if there are no other shared access capabilities in existence for this reference cell, meaning that the underlying representation can again be a pair of unsafe reference cells.

2.2.3 Regular protocols

What else can intuitionistic F$^\circ$ guarantee? Rather than present more individual examples, let us see how *any* protocol expressible as a regular language can be written in F$^\circ$. (This is not, of course, an upper limit on the expressivity of intuitionistic F$^\circ$—the previously seen RCRef and ShareRef are equivalent to the parenthesis matching that is the canonical non-regular example. The regular languages, however, still serve as rich space of example protocols that can be reasoned about in a straightforward way.)

As every regular language can be recognized by a deterministic finite automaton, let us 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 function from $\Sigma \times Q$ to Q (viewed also as a subset of $\Sigma \times Q \times Q$) describing the valid transitions between states, q_0 is a distinguished initial state, and F is a set of final states. Several of the examples seen previously can be seen as finite automata with alphabets of methods and capability parameters as. For example, UniqRef has two states, the second of which (Freed) is its only final state, GCRef is a trivial automaton with only exactly one (final) state (GC-managed), and, as the regular languages are closed under union and concatenation, augmenting UniqRef with the coercion gc yields a simple three state automaton with both of these final states.

To present the encoding of DFAs into F$^\circ$ as naturally as possible, let us extend the metavariable conventions used so far to allow q and r as type variables. Taking $Q = \{q_0, \ldots, q_n\}$, we can now define the automaton type τ_M for the DFA M as

$$\begin{aligned}\tau_M \triangleq \exists q_0{:}\circ, \ldots, q_n{:}\circ.\ q_0 \otimes \{\ & a\_\mathsf{takes}\_q\_\mathsf{to}\_q' : q \xrightarrow{\star} q' \\ & \quad \textit{for every } (a, q, q') \in \delta \\ & \vdots \\ & \mathsf{done\_at}\_q : q \xrightarrow{\star} \mathbf{1} \\ & \quad \textit{for every } q \in F\ \}\end{aligned}$$

Because the kind of q_0 is $\circ$, and because q_0 appears in the outermost tensor product of this construction, every τ_M must have kind $\circ$ as well; τ_M is also trivially inhabited for any M, as one can always supply an existential package in which all type variables are bound to $\mathbf{1}$ and every function is $\lambda^\star x{:}\mathbf{1}.\ x$.

If e contains some subexpression of type τ_M, let us say that the evaluation $e \longrightarrow^* v$ *reflects* a word $w = a_0 \dots a_k$ if the sequence of functions in that record that are applied during evaluation is exactly

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

for some states r_0 through r_k. To prove that τ_M is an accurate representation of M, it must be shown that each $w = a_0 \dots a_k$ accepted by M corresponds to an expression of type $\tau_M \xrightarrow{\star} \mathbf{1}$ that reflects w and that any f of type $\tau_M \xrightarrow{\star} \mathbf{1}$ reflects some w accepted by M. The former is fairly straightforward:

Lemma 3. *For any $w = a_0 \dots a_k$ accepted by M, there exists f such that $\cdot;\cdot \vdash f : \tau_M \xrightarrow{\star} \mathbf{1}$ and $f\ n$ reflects w for any closed n where $\cdot;\cdot \vdash n : \tau_M$.*

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 $r_k \in F$. From this and the definition of τ_M, we can construct

$$\begin{aligned}
f \ &: \ \tau_M \\
f \ &\triangleq \ \lambda^\star n{:}\tau_M.\ \mathbf{unpack}\ (q_0, \dots, q_n, p) = n\ \mathbf{in} \\
&\qquad \mathbf{let}\ (\mathit{init}, \mathit{trans}) = p\ \mathbf{in} \\
&\qquad \mathbf{let}\ s_0 = \mathit{trans}.a_0_\mathsf{takes}_q_0_\mathsf{to}_r_0\ \mathit{init}\ \mathbf{in} \\
&\qquad \vdots \\
&\qquad \mathbf{let}\ s_k = \mathit{trans}.a_k_\mathsf{takes}_r_{k-1}_\mathsf{to}_r_k\ s_{k-1}\ \mathbf{in} \\
&\qquad \mathit{trans}.\mathsf{done_at}_r_k\ s_k
\end{aligned}$$

Applying f to any closed n of type τ_M clearly reflects w, and, if w is in fact accepted by M, f will indeed have its declared type. □

The other direction depends on arguments justified by the nature of linearity, which will be examined in Section 2.3, and by parametricity, proved by Zhao et al. (2010).

Lemma 4. *If* $\cdot;\cdot \vdash f : \tau_M \xrightarrow{\star} \mathbf{1}$*, then there exists some* w *such that* M *accepts* w *and* $f\ n$ *reflects* w *for any* n *where* $\cdot;\cdot \vdash n : \tau_M$*.*

Proof. (Sketch.) Because τ_M is linear, f must eliminate n before it can return anything of type $\mathbf{1}$. This will require first unpacking the existential—as every τ_M is linear, this may only be done once—and then pattern-matching against the pair, leaving the linear initial state capability *init* and unrestricted operations record *trans*.

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

The above proof appeals to familiar intuitions about parametric polymorphism and the limited ways in which expressions whose types are variables can be used. The assurance that w is properly reflected, however, also depends on the connection between the static property having a linear rather than an unrestricted type and the behavior of expressions at runtime. Such a dependence is not unexpected—it is equally true that both lemmas depend on standard soundness' promise that the evaluation of well-typed expressions does not "go wrong". It does mean, however, that the discussion of soundness in the next section must make this connection between linearity and run-time behavior.

2.3 Metatheory

Soundness for intuitionistic F° implies that well typed terms cannot go wrong. As mentioned in the previous section, however, "not going wrong" must take into account the ways in which subexpressions of linear type may be used in addition to guaranteeing the absence of stuck terms. Happily, we shall see that standard sort of soundness for F°—*i.e.*, the ruling out of stuck terms—in fact implies that linear resources are used correctly.

2.3.1 Standard soundness

Progress and preservation make up the standard definition of type soundness. Progress, which states that a closed, well-typed expression that is not a value can always take an evaluation step, is no different than in ordinary System F:

Lemma 5 (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 and appeal to canonical forms—the property that values of a given type always have the shape of that type's introduction form. Completely standard. □

Preservation, which states that a well-typed expression that takes a step evaluates to another well-typed expressions, requires more care. As expected, it depends on various substitution lemmas, which state that a type of the correct kind may be substituted for a type variable bound by Γ in a kinding or typing derivation and that an expression of the correct type may be substituted for an expression variable bound by either Γ or Δ in a typing derivation.

Lemma 6 (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* $\Gamma; \Delta_1 \Cup \Delta' \Cup \Delta_2 \vdash \{x \mapsto e'\}e : \tau$*.*

Proof. Each proceeds by induction on the typing derivation of $e : \tau$ (or, in the first variant, the kinding derivation of $\tau : \kappa$). Weakening and contraction, mentioned in Section 2.1, are invoked in proving the third variant, while the fourth variant uses the $\Cup$ relation to ensure that the variables in Δ' do not overlap with those in Δ_1 or Δ_2. □

Note that the third substitution lemma types e' with respect to an empty linear context; it in fact does hold if e' is permitted to contain free linear variables. Similar results have been observed for other linear systems (Wadler, 1992; Maraist et al., 1995). Call-by-value reduction means that the expressions being substituted will always be values, however, and—because F$^\circ$ uses the value restriction—we can prove that unrestricted values contain no free linear variables:

Lemma 7. *If* $\Gamma; \Delta \vdash v : \tau$ *and* $\Gamma \vdash \tau : \star$, *then* $\Delta = \cdot$.

Proof. Simple induction on the typing derivation. □

Finally, a permutation property for linear contexts, stating that the order of variables is irrelevant, is required before preservation can be proved:

Lemma 8 (Permutation)**.** *If* $\Gamma; \Delta \vdash e : \tau$ *and* Δ' *is a permutation of* Δ, *then* $\Gamma; \Delta' \vdash e : \tau$.

Proof. Induction on the typing derivation and straightforward reasoning about permutations and the $\Cup$ relation. □

With these pieces in place, preservation follows easily:

Lemma 9 (Preservation)**.** *If* $\Gamma; \Delta \vdash e : \tau$ *and* $e \longrightarrow e'$, *then* $\Gamma; \Delta \vdash e' : \tau$.

Proof. Straightforward induction on the typing derivation and appeals to the various auxiliary lemmas given above. □

From preservation and progress, soundness follows naturally:

Theorem 10 (Type soundness)**.** *If* $\cdot; \cdot \vdash e{:}\tau$, *then for any* e' *where* $e \longrightarrow^* e'$, *either* $e' = v$ *for some* v *or there exists* e'' *such that* $e' \longrightarrow e''$.

Proof. Induction on $e \longrightarrow^* e'$ followed by direct application of Lemmas 5 and 9. □

Type soundness has been verified in Coq for the minimal subset of intuitionistic F$^\circ$ that contains only type variables, quantified types, and arrow types (Mazurak et al., 2010); aside from the additive sum and product, all the other constructs we have seen so far can be encoded in this sublanguage, and the additives present no additional complications.

A direct correspondence between these metatheoretic properties and intuitions about what linearity provides for the programmer is not immediately obvious, however. This is the topic of the next section.

2.3.2 Annotated semantics and linearity at run time

Formalize the intuitions about what linearity means for run-time behavior is not as straightforward as it might appear, as some common 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 run time, 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 seen previously involves a linear function, and we certainly want to allow for **let** expressions of unrestricted type.

Yet when we consider extending F° with anything that must be protected by linearity—*e.g.*, stateful constructs like the references cells presented in Section 2.2.2—we quickly find examples that, in a naive call-by-name or call-by-need variant of F°, would behave in ways that should be prevented. For example, given a reference cell as defined previously—of type UniqRef extended with gc—unpacked as $(\mathit{ref}, \mathit{methods})$, one might dispose of it and move on to evaluating e with

$$\mathit{methods}.\mathsf{gc}\ \mathit{ref}; e$$

This call to gc will only take place under call-by-value reduction—since its result is not used, other evaluation strategies would simply discard the subexpression, leaking memory that the programmer had supposed would be (eventually) reclaimed by the garbage collector. This is precisely the sort of error linearity should rule out rule out. (Clean, which is not call-by-value, grapples with concerns analogous to these in the context of its uniqueness types (Vries et al., 2008).)

Call-by-value reduction, with its assurance that function arguments are evaluated exactly once, seems well-suited to avoiding these problems; indeed, we shall see that a call-by-value setting—with the addition of the value restriction—rules out such problematic behavior. In other words, the restrictions already in place to ensure standard soundness

in the previous section are everything needed to ensure that linearity provides what we expect it will. To prove that this is so, the remainder of this section provides an extended operational semantics with which we can reason more formally about what linearity provides at run time.

Annotating intuitionistic F$^\circ$

What does it mean to "use" a subexpression? In other contexts, one might mean by this that an expression is evaluated to a value, that it is passed to a function, that it is applied to an argument, or something else entirely. In order to understand the run-time effects of linearity, which stem from restrictions on the use of variables, let us focus on two particular uses of values:

1. A value is *conscripted* when it replaces a variable in the body of a function, a **let** expression, or a **case** expression.

2. A previously conscripted value is *discharged* whenever evaluation proceeds *because* it is a value—that is, the syntax of F$^\circ$ may allow an arbitrary expression, but if that expression were not a value, evaluation would have continued in a different manner (typically by using evaluation contexts to decompose the expression further). In particular, had a free variable been present instead of the discharged value, evaluation would have been stuck.

Linearity's contribution is now clear: in the course of evaluation, every conscripted linear value must be discharged exactly once—unless it is suspended within the final result, in which case it must not be discharged. Focusing exclusively on conscripted values captures the fact that linearity is, at its core, about variables: until a value is substituted for a variable, the question of whether this value is "treated linearly" is meaningless.

This can be formalized by defining a linearity-aware operational semantics on expressions extended with values that are tagged with a type and *identifier*—any unique token will suffice, as identifiers serve merely to match uses of a value corresponding to uses of a variable, so let us use natural numbers j, k, and ℓ. Tagged values are those that have been conscripted; for reasons that will become clear shortly, tagged types, type annotations on

$$[\text{L-App}]\ (\lambda^{\kappa} x{:}\tau.\ e)\ v \xrightarrow[\epsilon]{(v:\tau)^j}{}^{j}_{j+1}\ \{x \mapsto (v{:}\tau)^j\}e \qquad [\text{L-Select}]\ \langle e_1, e_2\rangle.i \xrightarrow[\epsilon]{\epsilon}{}^{j}_{j}\ e_i$$

$$[\text{L-TApp}]\ \Lambda\alpha{:}\kappa.\ v\ [\tau] \xrightarrow[\epsilon]{\epsilon}{}^{j}_{j}\ \{\alpha \mapsto (\tau{:}\kappa)\}v \qquad [\text{L-Seq}]\ ();e \xrightarrow[\epsilon]{\epsilon}{}^{j}_{j}\ e$$

$$[\text{L-Let}]\ \begin{array}{c}\textbf{let}^{\tau_1 \otimes \tau_2}\ (x_1, x_2) = (v_1, v_2)\ \textbf{in}\ e \xrightarrow[\epsilon]{(v_1:\tau_1)^j,\,(v_2:\tau_2)^{j+1}}{}^{j}_{j+2} \\ \{x_1 \mapsto (v_1{:}\tau_1)^j, x_2 \mapsto (v_2{:}\tau_2)^{j+1}\}e\end{array}$$

$$[\text{L-Case}]\ \textbf{case}\ \textbf{inj}_i^{\tau_1 \oplus \tau_2}\ v\ \textbf{of}\ \textbf{inj}_1\ x_1 \mapsto e_1 \mid \textbf{inj}_2\ x_2 \mapsto e_2 \xrightarrow[\epsilon]{(v:\tau_i)^j}{}^{j}_{j+1}\ \{x_i \mapsto (v{:}\tau)^j\}e_i$$

$$[\text{L-Unpack}]\ \begin{array}{c}\textbf{unpack}\ (\alpha, x) = (\textbf{pack}\ \alpha{:}\kappa = \tau\ \textbf{in}\ (v : \tau'))\ \textbf{in}\ e \xrightarrow[\epsilon]{(v:\tau')^j}{}^{j}_{j+1} \\ \{\alpha \mapsto (\tau{:}\kappa)^{\alpha} x \mapsto (v{:}\tau')^j\}e\end{array}$$

$$[\text{L-Tag}]\ (v{:}\tau)^k \xrightarrow[(v:\tau)^k]{\epsilon}{}^{j}_{j}\ v \qquad [\text{L-Ctxt}]\ \frac{e \xrightarrow[D]{C}{}^{j}_{k}\ e'}{E\big[e\big] \xrightarrow[D]{C}{}^{j}_{k}\ E\big[e'\big]}$$

$$[\text{L}^*\text{-Id}]\ e \xrightarrow[\epsilon]{\epsilon}{}^{*\,j}_{j}\ e \qquad [\text{L}^*\text{-Trans}]\ \frac{e \xrightarrow[D_1]{C_1}{}^{j}_{k}\ e' \qquad e' \xrightarrow[D_2]{C_2}{}^{*\,k}_{\ell}\ e''}{e \xrightarrow[D_1,\,D_2]{C_1,\,C_2}{}^{*\,j}_{\ell}\ e''}$$

Figure 2.6: Linearity-aware semantics for intuitionistic F°

let expressions, and sequences of tagged values are also required:

$$\begin{array}{rcl} j, k, \ell & ::= & 0 \mid 1 \mid 2 \mid \ldots \\ \\ \tau & ::= & \ldots \mid (\tau{:}\kappa) \\ e & ::= & \ldots \mid (v{:}\tau)^j \mid \textbf{let}^{\tau \otimes \tau}\ (x, x) = e\ \textbf{in}\ e \\ \\ C, D & ::= & \epsilon \mid (v{:}\tau)^j \mid C, C \end{array}$$

An augmented semantics for reasoning about this annotated variant of intuitionistic F° is given in Figure 2.6: $e \xrightarrow[D]{C}{}^{j}_{k}\ e'$ means that, given $j > \ell$ for every identifier ℓ in e, e steps to e' while conscripting the sequence of tagged values C and discharging the sequence D—the identifier $k \geq j$ is guaranteed not to appear in e', C, or D. Values are conscripted in rules L-AppLam, L-Let, L-Case, and L-Unpack; *i.e.*, whenever a value would be substituted into an expression, it is first tagged with the variable it is replacing, along with

$$[\text{K-TTAG}]\ \frac{\Gamma \vdash \tau : \kappa}{\Gamma \vdash (\tau{:}\kappa) : \kappa} \qquad [\text{T-TAG}]\ \frac{\Gamma \vdash v : \tau}{\Gamma \vdash (v{:}\tau)^j : \tau} \qquad [\text{T-TTAG1}]\ \frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash \tau : \kappa}{\Gamma \vdash e : (\tau{:}\kappa)}$$

$$[\text{T-TTAG2}]\ \frac{\Gamma \vdash e : (\tau{:}\kappa)}{\Gamma \vdash e : \tau}$$

Figure 2.7: New kinding and typing rules for annotated intuitionistic F°

that variable's type. Similarly, while L-TAPPLAM does not conscript, it does tag its type argument with its expected kind, as does L-UNPACK; this keeps a record of the kind at which that type will be considered within its body (and requires no identifier).

Since a tagged value $(v{:}\tau)^j$ is not itself a value—a necessary constraint to keep evaluation deterministic—rule L-TAG discharges the tagged value by removing its tag. Between being conscripted and discharged, a tagged value may linger within a function closure or additive pair, and a tagged value suspended within the final result of evaluation will never be discharged. As before, evaluation does not depend on typing, but the record kept by C and D in $e \xrightarrow[D]{C} {}^{*\,j}_{k}\ e'$ does reveal how arguments considered at various types were used.

The augmented kinding and typing rules in Figure 2.7 are straightforward: the rules K-TTAG and T-TAG simply check that the tagged type or expression can indeed have the kind or type that the tag declares it has. The rules T-TTAG1 and T-TTAG2 make kind annotations completely transparent at the expression level.

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

$$\begin{array}{lll}
(\Lambda\alpha{:}\circ.\ \lambda^\circ x{:}\alpha.\ x)\ [\mathsf{Nat}]\ 42 & \xrightarrow[\epsilon]{\epsilon}{}^{0}_{0} & (\lambda^\circ x{:}(\mathsf{Nat}{:}\circ).\ x)\ 42 \quad \text{L-TAPPLAM} \\
 & \xrightarrow[\epsilon]{(42:(\mathsf{Nat}:\circ))^0}{}^{0}_{1} & (42{:}(\mathsf{Nat}{:}\circ))^0 \quad \text{L-APPLAM} \\
 & \xrightarrow[(42:(\mathsf{Nat}:\circ))^0]{\epsilon}{}^{1}_{1} & 42 \quad \text{L-TAG}
\end{array}$$

Or, more concisely:

$$(\Lambda\alpha{:}\circ.\ \lambda^\circ x{:}\alpha.\ x)\ [\mathsf{Nat}]\ 42 \xrightarrow[(42:(\mathsf{Nat}:\circ))^0]{(42:(\mathsf{Nat}:\circ))^0}{}^{*\,0}_{1}\ 42$$

In other words: over the course of evaluation, the value 42 was passed to a function and used (in this case, returned) exactly once, and it was considered at type $(\mathsf{Nat}{:}\circ)$. Were this

$$
\begin{array}{rcl}
\mathfrak{T}((v{:}\tau)^j) & = & \{(v{:}\tau)^j\} \uplus \mathfrak{T}(v) \\
\mathfrak{T}(x) & = & \emptyset \\
\mathfrak{T}(\lambda^\kappa x{:}\sigma_1.\, e) & = & \mathfrak{T}(e) \\
\mathfrak{T}(e_1\ e_2) & = & \mathfrak{T}(e_1) \uplus \mathfrak{T}(e_2) \\
\mathfrak{T}(\langle e_1, e_2\rangle) & = & \mathfrak{T}(e_1) \cup \mathfrak{T}(e_2) \\
\mathfrak{T}(e.i) & = & \mathfrak{T}(e) \\
\mathfrak{T}(\Lambda\alpha{:}\kappa.\, v) & = & \mathfrak{T}(v) \\
\mathfrak{T}(e\ [\tau]) & = & \mathfrak{T}(e) \\
\mathfrak{T}(()) & = & \emptyset \\
\mathfrak{T}(e_1; e_2) & = & \mathfrak{T}(e_1) \uplus \mathfrak{T}(e_2) \\
\mathfrak{T}((e_1, e_2)) & = & \mathfrak{T}(e_1) \uplus \mathfrak{T}(e_2) \\
\mathfrak{T}(\textbf{let}\ (x_1, x_2) = e_1\ \textbf{in}\ e_2) & = & \mathfrak{T}(e_1) \uplus \mathfrak{T}(e_2) \\
\mathfrak{T}(\textbf{inj}_i^{\tau_1 \oplus \tau_2}\ e) & = & \mathfrak{T}(e) \\
\mathfrak{T}(\textbf{case}\ e\ \textbf{of}\ \textbf{inj}_1\ x_1 \mapsto e_1 \mid \textbf{inj}_2\ x_2 \mapsto e_2) & = & \mathfrak{T}(e) \uplus (\mathfrak{T}(e_1) \cup \mathfrak{T}(e_2)) \\
\mathfrak{T}(\textbf{pack}\ \alpha{:}\kappa = \tau\ \textbf{in}\ (e : \tau')) & = & \mathfrak{T}(e) \\
\mathfrak{T}(\textbf{unpack}\ (\alpha, x) = e_1\ \textbf{in}\ e_2) & = & \mathfrak{T}(e_1) \uplus \mathfrak{T}(e_2)
\end{array}
$$

Figure 2.8: Tagged contents of intuitionistic F° expressions

tagged 42 to appear more than once in the list of discharged values, linearity would have failed us—although 42 is of type Nat, here it is being considered as linear because α was declared to be linear; conscripted at this type, it should be discharged exactly once. Were it to instead appear as $(42{:}\mathsf{Nat})^0$ or $(42{:}(\alpha{:}\star))^0$, however, there should be no such restriction on its use, as one can indeed write functions from Nat to Nat that use their arguments more than once or that ignore them altogether.

This annotated variant of intuitionistic F° is reminiscent of linear languages with heap semantics (Turner and Wadler, 1999), which involve tracking, instead of the traces along the lines of C and D, a single heap H. Such languages often perform substitution when the expression being substituted is unrestricted, while associating that expression with the variable in the heap in the linear case. Theorem 12 could still be proved with such a semantics, but the ability to reason about the order of substitutions is necessary to support the reasoning used in Section 2.2.3.

Linearity at run time

A few more auxiliary definitions are necessary before the connection between linearity of types and run-time behavior can be verified. The multiset of tagged values appearing

within e is written $\mathfrak{T}(e)$ and defined in Figure 2.8. A sequence C treated as a multiset is written $\{C\}$, and, if S is a multiset of tagged values, $S\backslash\kappa$ is the subset of S that omits any $(v{:}\tau)^j$ where $\cdot \vdash \tau : \kappa$.

Finally, let us call an expression *proper* if it contains no unrestricted value which contains contains a value tagged with a 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 $\mathfrak{T}(v)\backslash\star = \emptyset$. Every unannotated F° term is trivially proper. This property is preserved by evaluation:

Lemma 11 (Proper expressions)**.** *If e is proper and $e \overset{C}{\underset{D}{\longrightarrow}}{}^{j}_{k}\ e'$, then e' is also proper.*

Proof. 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 within an unrestricted value that suspends computation. This is prevented for all such values: rule T-LAM requires an empty linear context for unrestricted functions, the additive product type is never unrestricted, and the value restriction rules out type abstractions. □

It is finally possible to prove that linearity guarantees a correspondence between values conscripted and discharged at linear type:

Theorem 12 (Run time linearity)**.** *If $\Gamma; \Delta \vdash e : \tau$, e is proper, and $e \overset{C}{\underset{D}{\longrightarrow}}{}^{j}_{k}\ e'$ for some e', C, D, j, and k, then $\mathfrak{T}(e)\backslash\star \uplus \{C\}\backslash\star = \mathfrak{T}(e')\backslash\star \uplus \{D\}\backslash\star$.*

Proof. Straightforward induction over the annotated evaluation relation. L-APP, L-LET, L-CASE, and L-UNPACK are the only interesting cases, and they all proceed the same way: from T-UVAR and Lemma 11 we know that, unless the substituted value occurs exactly once, it will be of unrestricted type and contain no linear tags. The balance of tags is thus preserved. □

In other words, at run time, the values that are substituted for variables of linear type are never duplicated or discarded.

2.4 Related work

There are many linear type systems and variants thereof in the literature (Walker, 2005), and intuitionistic linear systems outnumber classical systems significantly, in part for reasons that will become more clear in Chapter 3. The use of kinds and kind subsumption in F$^\circ$ is intended to capture the essence of linearity simply and generally, retaining a semantics and programming model from System F, and is most similar in this regard to the monomorphic systems of Wadler (1990) and Benton (1995). Alms (Tov and Pucella, 2011a) has adapted this approach to *affine* types, which permit weakening but not contraction, and adds some additional features. Rust (Hoare, 2010), while not explicitly based on linear or affine logic, requires that types implement a `Copy` trait if members of this type may safely be duplicated—basic traits like `Copy` are even referred to as kinds.

Other linear systems that admit some unrestricted behavior have used different mechanisms for distinguishing between the linear and the unrestricted, and the interactions between these mechanisms and polymorphism can differ significantly. Broadly speaking, there are two other approaches taken to divide linear and unrestricted variables: following the lead of intuitionistic linear logic or using type qualifiers.

2.4.1 Intuitionistic linear logic

Following linear logic as originally presented (Girard, 1987) leads to treating *all* types as linear by default and introducing the modal constructor ! to allow for unrestricted terms, which must be closed with respect to linear variables. A polymorphic λ-calculus with this form of linearity (and metavariables distinct from those used for F$^\circ$) might look like

$$\sigma \quad ::= \quad \alpha \mid \sigma \multimap \sigma \mid \forall\alpha.\, \sigma \mid !\sigma \qquad \textit{types}$$

$$t \quad ::= \quad a \mid x \mid \lambda a{:}\sigma.\, t \mid t\ t \mid \Lambda\alpha.\, t \mid t\ [\sigma] \mid !t \mid \textbf{let}\ !x = t\ \textbf{in}\ t \qquad \textit{terms}$$

$$\Phi \quad ::= \quad \cdot \mid \Phi, \alpha \mid \Phi, x{:}\sigma \qquad \textit{unrestricted typing contexts}$$

$$\Psi \quad ::= \quad \cdot \mid \Psi, a{:}\sigma \qquad \textit{linear typing contexts}$$

Such languages generally distinguish between linear variables a, bound by λ terms, and non-linear variables x, bound by **let** !, the elimination form of !. It is common for ! to

suspend computation (Maraist et al., 1995), and in any case **let** ! evaluates as expected:

$$\textbf{let}\ !x = !t\ \textbf{in}\ t' \longrightarrow \{x \mapsto t\}t'$$

As all types in such a language are linear, $\sigma_1 \multimap \sigma_2$ is and $\forall\alpha.\ \sigma$ behave exactly like $\tau_1 \stackrel{\circ}{\to} \tau_2$ and $\forall\alpha{:}\circ.\ \tau$ in F$^\circ$. The typing rules concerning the ! modality demonstrate how unrestricted behavior is controlled:

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

That is, the type $!\sigma$ indicates a term of type σ that uses no linear variables—the same constraint F$^\circ$ places on unrestricted functions—and such a term can be captured by an unrestricted variable, and thus duplicated or discarded, using the **let** !. Note that one can still have linear variables type $!\sigma$, 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 2.3, F$^\circ$ requires call-by-value reduction and the value restriction—and sound formulations end up heavier than those that make this distinction (Wadler, 1993; Benton et al., 1993; Ahmed et al., 2007).

The above system can be encoded in intuitionistic F$^\circ$ easily enough, starting with a translation on types $\lceil\lceil\sigma\rceil\rceil$

$$\begin{aligned}
\lceil\lceil\alpha\rceil\rceil &= \alpha \\
\lceil\lceil\sigma_1 \multimap \sigma_2\rceil\rceil &= \lceil\lceil\sigma_1\rceil\rceil \stackrel{\circ}{\to} \lceil\lceil\sigma_2\rceil\rceil \\
\lceil\lceil\forall\alpha.\ \sigma\rceil\rceil &= \forall\alpha{:}\circ.\ \lceil\lceil\sigma\rceil\rceil \\
\lceil\lceil!\sigma\rceil\rceil &= \mathbf{1} \stackrel{\star}{\to} \lceil\lceil\sigma\rceil\rceil
\end{aligned}$$

The corresponding translation on terms is straightforward given the translation on types.[4] The only interesting cases involve unrestricted variables and the ! modality; the rest map $\lceil\lceil -\rceil\rceil$ homomorphically over the term:

$$\begin{aligned}
\lceil\lceil x\rceil\rceil &= x\ () \\
\lceil\lceil !t\rceil\rceil &= \lambda^\star _{:}\mathbf{1}.\ \lceil\lceil t\rceil\rceil \\
\lceil\lceil \textbf{let}\ !x = t_1\ \textbf{in}\ t_2\rceil\rceil &= (\lambda^\circ x{:}\mathbf{1} \stackrel{\star}{\to} \lceil\lceil\sigma_1\rceil\rceil.\ \lceil\lceil t_2\rceil\rceil)\ \lceil\lceil t_1\rceil\rceil \\
&\qquad \textit{where } t_1 \textit{ has type } !\sigma_1
\end{aligned}$$

[4] The translation is technically defined on typing derivations, as is evident from the translation of **let** !.

This style of linearity works well with polymorphism—as all types are treated uniformly, any type can be substituted for any type variable—and has seen much use in the literature (Wadler, 1992; Barber, 1997; Bierman et al., 2000; Wadler, 1992). Unfortunately, limiting unrestricted terms to suspended computations that depend on no linear variables can make programming quite cumbersome, and it contradicts the programmer's intuition that unrestricted values are simply ordinary values that may be used in less restricted fashions—with !, every unrestricted value must essentially be created and used as a thunk. Indeed, unrestrictedness is the norm in most programs rather than the exception, and ! is cumbersome even at base types—the constant 3 would be of type Int, not !Int, and would thus need to be bound by **let** ! before it could be duplicated. One might hope to write in a cleaner surface language and infer all !s and **let** !s, but such inference has been shown to be undecidable in the general case (Lincoln and Mitchell, 1992).

2.4.2 Type qualifiers

The second approach to distinguishing linear from unrestricted types uses *qualifiers* lin (for "linear") and un (for "unrestricted") applied to a collection of pre-types (Ahmed et al., 2005; Walker, 2005; Tov and Pucella, 2011b). These qualifiers constrain usage, 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—in particular, it raises the question of whether quantification should be over types, pre-types, or qualifiers, and for maximum expressivity a language must support all three. This quickly leads to large and complex types; for instance, the type of plus might have as many as 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 these qualifiers are nontrivial—*e.g.*, q_5 must be lin if q_1 is—which can be captured, at the expense of additional complexity, by qualifier-level bounded quantification.[5] Qualifiers thus ease the use of unrestricted types but are too unwieldy for a

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

polymorphic source language—in fact, Wansbrough and Peyton Jones (1999) argue against similar qualifiers even in an intermediate language.

Systems with qualifiers often feature more than just lin and un, as *relevant* types (which allow contraction but not weakening) and affine types fit naturally into the qualifier lattice between linear and unrestricted types—the ways in which this lattice can be used, however, depend on what substructural types are used for in the language in question. Ahmed et al. (2005) and Tov and Pucella (2011b), for instance, use very similar languages for notably different purposes. Clean's uniqueness types (Hicks et al., 2004; Vries et al., 2008) can also be seen as a limited form of qualified types—uniqueness differs from linearity, however, as it refers more to the past ("Has uniqueness been preserved up until now?") than to the future ("Will linearity be respected?").

The use of kinds in lieu of either ! or type qualifiers appears to capture the strengths of both approaches. 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, kind $\circ$), while the type of plus is the simple $\mathsf{Int} \xrightarrow{\star} \mathsf{Int} \xrightarrow{\star} \mathsf{Int}$. F$^\circ$ thus has flexible polymorphic types without the need for bounded quantification or other complexities of subtyping in a higher-order setting. It also makes linearity a property inherent to certain types rather than a property of assumptions or qualifiers, which fits the idea that certain types are naturally linear or unrestricted. All of this makes intuitionistic F$^\circ$ well suited to extension into a classical setting with a focus on concurrency, the topic of the next chapter.

Chapter 3

Classical F$^\circ$

Let us now consider its extension to a classical setting, in which processes and interprocess communication emerge quite naturally. Classical F$^\circ$ provides interprocess communication without unexpected message errors, without resource leaks, and without deadlock—this last property being one that requires substantially more machinery in languages that start with a more general process model (Kobayashi, 2002). Before presenting classical F$^\circ$, however, it is worth looking at what other classical linear systems have done, as this will highlight the novel aspects of classical F$^\circ$'s design, which keep classical F$^\circ$ close to what is already familiar in typed functional programming.

The differences between ordinary intuitionistic and classical logic can be seen in their treatment of negation and disjunction, and the same is true for linear logics. Traditional presentations of classical linear logic define negation via a *dualizing operator* $(-)^\perp$, suggesting the connection between negation and $\perp$, a more well-behaved linear representation of falsity.[1] Dualization pushes negation through a proposition (or type) in a manner reminiscent of de Morgan's laws, so in these languages one might see

$$\begin{array}{rcl@{\qquad}rcl}
\perp^\perp & \triangleq & \mathbf{1} & \mathbf{1}^\perp & \triangleq & \perp \\
(\sigma_1 \,\&\, \sigma_2)^\perp & \triangleq & \sigma_1^\perp \oplus \sigma_2^\perp & (\sigma_1 \oplus \sigma_2)^\perp & \triangleq & \sigma_1^\perp \,\&\, \sigma_2^\perp \\
(\sigma_1 \parr \sigma_2)^\perp & \triangleq & \sigma_1^\perp \otimes \sigma_2^\perp & (\sigma_1 \otimes \sigma_2)^\perp & \triangleq & \sigma_1^\perp \parr \sigma_2^\perp \\
\forall\alpha.\, \sigma^\perp & \triangleq & \exists\alpha.\, \sigma^\perp & \exists\alpha.\, \sigma^\perp & \triangleq & \forall\alpha.\, \sigma^\perp
\end{array}$$

[1] In particular, it is *not* true that everything follows from $\perp$; some linear systems do have such an "additive" false, written usually written 0, that has this property.

This dualization is clearly an involution—that is, $(\sigma^{\perp})^{\perp} = \sigma$—satisfying the requirement that a classical logic support double negation elimination. Classical logics also typically define implication in terms of disjunction, and indeed $\multimap$ is absent from the above definition. Linear implication is clearly one of linear logic's questionably termed *multiplicative* connectives, however—like $\otimes$, it combines and separates sets of linear assumptions—so it should not be defined in terms of the additive disjunction $\oplus$. Instead, $\multimap$ is defined in terms of a construct new to classical linear logic, a multiplicative disjunction $\parr$, pronounced "par":

$$\sigma_1 \multimap \sigma_2 \quad \triangleq \quad \sigma_1^{\perp} \parr \sigma_2^{\perp}$$

Classical linear logic is typically formulated as a sequent calculus—having as many connectives as it does, the fact that the introduction of σ is equivalent to the elimination of $\sigma^{\perp}$ represents a substantial reduction in the number of rules that must be written—and $\parr$ is a natural fit in such a setting, internalizing the comma on the right side of a typing judgment just as $\otimes$ internalizes a comma on the left.[2]

Yet, as convenient as sequent calculi are for proof theory, F$^{\circ}$ aims to suggest modest (if powerful) extensions to existing functional programming languages, and term languages for sequent calculi (*e.g.*, the dual calculus (Wadler, 2003)) do not fit the bill. In a natural deduction style language like F$^{\circ}$, it is not immediately clear how to interpret $\parr$—in fact, the resource-based intuitions commonly invoked for linear connectives ($\otimes$ is possession of both, $\&$ is the ability to make either, $\oplus$ is already having either one or the other) do not suggest an intuition for $\parr$. Yet there are tantalizing glimpses of concurrency within classical linear logic—Girard (1987) even refers to the components of $\parr$ as communicating.

For these reasons, classical F$^{\circ}$—which is not defined as a sequent calculus—forgoes $\parr$ in favor of arrow types, taking the intuitionistic interpretation of negation as implying false. This leads to double negations that must be explicitly eliminated, the task of one of the (relatively few) constructs in classical F$^{\circ}$ that were not present in intuitionistic F$^{\circ}$. These constructs are perhaps best introduced by beginning with the control operators of Felleisen and Hieb (1992), whose types in an unrestricted setting show a connection to

[2] Recall that the restriction to a single proposition on the right makes a sequent calculus intuitionistic rather than classical.

classical logic (Griffin, 1990; Ong and Stewart, 1997), and considering how they might be made to cooperate with linearity.

3.1 Motivations, terms, and types

The operators **abort** and **control** (a cousin of **callcc**) enable many interesting behaviors in a solely unrestricted setting (Felleisen and Hieb, 1992). Their behavior is easily defined using evaluation contexts:

$$\begin{aligned} E\big[\textbf{control}\ v\big] &\longrightarrow (v\ (\lambda x.\ \textbf{abort}\ E\big[x\big]) \\ E\big[\textbf{abort}\ v\big] &\longrightarrow v \end{aligned}$$

That is, **abort** discards the current evaluation context and returns its argument, while **control** supplies v—presumed to be a function that accepts a *continuation* argument—with a function that uses **abort** to return to the evaluation context E in which **control** appeared. Unlike **callcc**, **control** does not duplicate that evaluation context, making it potentially more compatible with linearity; **abort**, however, clearly has no place in a linear system, as it discards an evaluation context, which might well contain linear resources.

Let us consider how **control** might be redefined without **abort**, so that it could be used in a linear setting: invoking the continuation passed to v can be seen as returning an "answer" to the context E, and, if v is linear, this must eventually happen. Rather than hide E under a continuation function, then, why not conceive of it as another process, waiting for a response? Writing $\lfloor a \rfloor$ and $\lceil a \rceil$ respectively for the *sink* (*i.e.*, sending) and *source* (*i.e.*, receiving) endpoints of a channel a, a first attempt at reconciling **control** with a linear system might forgo **abort** and instead evaluate to a parallel composition of expressions:

$$E\big[\textbf{control}\ v\big] \longrightarrow E\big[\textbf{control}\ \lceil a \rceil\big] \mid v\ \lfloor a \rfloor$$

Evaluating **control** v now spawns v as a child process; the endpoints of the channel a link the parent and child processes, with the sink endpoint taking the place of the continuation function and the source endpoint standing in for v in the parent process. Evaluation of $v\ \lfloor a \rfloor$ can now proceed until $\lfloor a \rfloor$ is applied, at which point the argument to this application can be passed back to the parent process:

$$E\big[\textbf{control}\ \lceil a \rceil\big] \mid E'\big[\lfloor a \rfloor\ v\big] \longrightarrow E\big[v\big] \mid E'\big[\lfloor a \rfloor\big]$$

The closed channel $\rfloor a \lfloor$ indicates that communication over a is finished; it also allows the child process to terminate, but, before process termination actually occurs, all linear resources in E' must be safely consumed:

$$e \mid \rfloor a \lfloor \quad \longrightarrow \quad e$$

All these operations respect linearity, as neither expressions nor evaluation contexts are duplicated or discarded. Further, as long as the endpoints are linear, the eventual reply across the channel will always take place.

Yet **control**, as described above, offers a very poor form of concurrency—the parent process immediately blocks waiting for the child process to return. For actual parallel execution, classical F$^\circ$ splits the functionality of **control** across two operations. The first, **go**, spawns the child process and passes it the sink endpoint of a newly created channel, returning the source endpoint to the parent; with type-related details still elided, the previous example becomes

$$E\big[\mathbf{go}\ v\big] \quad \longrightarrow \quad E\big[\uparrow a \uparrow\big] \mid v \downarrow a \downarrow$$

The second operation, **yield**, blocks on a source endpoint until the corresponding sink endpoint is used. While again eliding types and following the running example, **yield** behaves as

$$E\big[\mathbf{yield}\ \uparrow a \uparrow\big] \mid E'\big[\downarrow a \downarrow v\big] \quad \longrightarrow \quad E\big[v\big] \mid E'\big[\rfloor a \lfloor\big]$$

Perhaps surprisingly, this is nearly all that needs to be added to F$^\circ$ in moving from an intuitionistic to a classical (and concurrent) setting, though the examples given so far have been simplified slightly. Figure 3.1 gives the full list of new syntax: in addition to **go** and **yield**, classical F$^\circ$ requires the type $\perp$, potentially negated type variables ω, a strictly linear kind $\bullet$, and channel typing contexts Π. The processes P that expressions may evaluate into are very simple, featuring only parallel composition $P_1 \mid P_2$ and channel binding $\nu a{:}\rho.\ P$, both with syntax taken from the π-calculus (Milner et al., 1992).

The full extent of these new constructs' functionality, however, has not yet been shown. It is most instructive to demonstrate this in conjunction with these new constructs' typing rules, as their logical and operational interpretations inform each other significantly.

$$
\begin{array}{rcll}
\omega & ::= & \alpha \mid \widetilde{\omega} & \textit{potentially negated type variables} \\
\kappa & ::= & \ldots \mid \bullet & \textit{new kinds} \\
\tau, \rho & ::= & \ldots \mid \omega \mid \bot & \textit{new types} \\
e & ::= & \ldots \mid \mathbf{go}^{\rho}\ e \mid \mathbf{yield}\ e \mid \uparrow a \uparrow \mid \downarrow a \downarrow \mid \lfloor a \rfloor & \textit{new expressions} \\
\\
v & ::= & \ldots \mid \uparrow a \uparrow \mid \downarrow a \downarrow \mid \lfloor a \rfloor & \textit{new values} \\
E & ::= & \ldots \mid \mathbf{go}^{\rho}\ E \mid \mathbf{yield}\ E & \textit{new evaluation contexts} \\
\\
P & ::= & e \mid P \mid P \mid \nu a{:}\rho.\ P & \textit{processes} \\
\\
\Pi & ::= & \cdot \mid \Pi, a{\cdot}\rho \mid \Pi, a{\tilde{\cdot}}\rho \mid \Pi, a{:}\rho & \textit{channel contexts}
\end{array}
$$

Figure 3.1: New syntax for classical F°

The kind $\bullet$ classifies those types that can serve as the type annotations on channels, appearing in ν binders and channel contexts Π. The examples presented so far use only two types in this manner, $\bot$ and $\overset{\bullet}{\to}$. Types of the form $(\tau \overset{\bullet}{\to} \bot) \overset{\bullet}{\to} \bot$ occur frequently, so let us abbreviate such a type as $\uparrow\tau\uparrow$, pronounced "source of τ".

As this notation suggests, **yield** mediates between the types $\uparrow\tau\uparrow$ and τ—in other words, it provides double negation elimination, making its typing straightforward:

$$
[\text{T-YIELD}]\ \frac{\Gamma; \Delta; \Pi \vdash e : \uparrow\tau\uparrow}{\Gamma; \Delta; \Pi \vdash \mathbf{yield}\ e : \tau}
$$

A double negation elimination means more than simply receiving a value, however; to see how much more, let us turn to **go**.

The earlier examples portrayed **go** as spawning processes that simply compute values and return them. A child process that communicates by returning a single value of type τ before exiting requires a channel sink endpoint of type $\tau \overset{\bullet}{\to} \bot$, so let us consider $\tau \overset{\bullet}{\to} \bot$ to be the *protocol type* of such a channel. The kind $\bullet$ classifies exactly these protocol types, which by convention are written ρ. In fact, **go** can spawn processes with *any* protocol type: its argument must have the type $\rho \overset{\bullet}{\to} \bot$ for some ρ; it will be given a sink endpoint $\downarrow a \downarrow$ of type ρ and eventually terminate with a closed channel $\lfloor a \rfloor$ of type $\bot$. (Because **go** must also create the ν binder for a, which includes a type annotation, it is in fact written $\mathbf{go}^{\rho}$ in

$$[\text{K-ANSWER}]\ \Gamma \vdash \perp\!\!\!\perp : \bullet \qquad\qquad [\text{K-DUAL}]\ \frac{\Gamma \vdash \omega : \bullet}{\Gamma \vdash \widetilde{\omega} : \bullet}$$

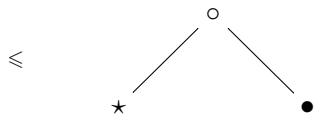
$$[\text{K-ARROW}]\ \frac{\Gamma \vdash \tau_1 : \kappa_1 \qquad \Gamma \vdash \tau_2 : \kappa_2 \qquad \kappa = \bullet \implies \kappa_2 = \bullet}{\Gamma \vdash \tau_1 \xrightarrow{\kappa} \tau_2 : \kappa}$$

$$[\text{K-WITH}']\ \frac{\Gamma \vdash \tau_1 : \bullet \qquad \Gamma \vdash \tau_2 : \bullet}{\Gamma \vdash \tau_1 \,\&\, \tau_2 : \bullet}$$

Figure 3.2: New and modified kinding rules for classical F°

keeping with the convention established for additive sum injection.)

Seeing protocol types as the types of sink endpoints, which types make sense as protocols? A protocol might specify that a value of type τ is to be sent before continuing according to rho, the rest of the protocol, as has been demonstrated ($\tau \xrightarrow{\bullet} \rho$); it might specify that a choice between protocols ρ_1 and ρ_2 is to be made ($\rho_1 \,\&\, \rho_2$); it might specify that a type of kind κ is to be sent over the channel ($\forall \alpha{:}\kappa.\ \rho$); or it might be complete ($\perp\!\!\!\perp$).[3] The revised kinding rules in Figure 3.2, along with those already presented in Figure 2.3, allow exactly these types to be classified by $\bullet$—which serves to both restrict the shapes of types which may be considered as protocols and to provide a linear kind that, unlike $\circ$, cannot also classify unrestricted types—and the subkinding order is as would be expected:



Source endpoints need types as well. As **yield** will be applied to these endpoints, the type of a source endpoint should be doubly negated. If $\downarrow a \downarrow$ has type ρ, the type of $\uparrow a \uparrow$ is the

[3] In linear logic, the protocol type connectives are said to be *negative*, meaning that their introduction forms are invertible. That is, no choice is encoded in their construction—in contrast to the choice of injection for $\oplus$, the choice of resource split for $\otimes$, and the choice encapsulated type for $\exists$, all of which are considered *positive* connectives.

doubly negated dual $\widetilde{\rho}$, defined as follows:

$$\begin{aligned}
\widetilde{\tau \overset{\bullet}{\to} \rho} &\triangleq \uparrow \tau \otimes \widetilde{\rho} \uparrow \\
\widetilde{\rho_1 \,\&\, \rho_2} &\triangleq \uparrow \widetilde{\rho_1} \oplus \widetilde{\rho_2} \uparrow \\
\widetilde{\forall \alpha{:}\kappa.\ \rho} &\triangleq \uparrow \exists \alpha{:}\kappa.\ \widetilde{\rho} \uparrow \\
\widetilde{\bot} &\triangleq \uparrow \mathbf{1} \uparrow \\
\widetilde{\omega} &\triangleq \widetilde{\omega}
\end{aligned}$$

For α of kind $\bullet$ there can be no further computation of $\widetilde{\alpha}$ until another type is substituted for α, and similarly for $\widetilde{\widetilde{\alpha}}$—$\omega$ represents a type variable that has been dualized an arbitrary number of times. The definition of substitution must be extended as this syntax suggests, so that such suspended dualizations are computed immediately:

$$\begin{aligned}
\{\alpha \mapsto \rho\}\widetilde{\alpha} &= \widetilde{\rho} \\
\{\alpha \mapsto \rho\}\widetilde{\widetilde{\alpha}} &= \widetilde{\widetilde{\rho}} \\
&\vdots
\end{aligned}$$

This dualization is exactly the left-hand column in the definition of $(-)^{\bot}$ given at the beginning of the chapter, excepting that $\parr$ has been replaced by $\overset{\bullet}{\to}$ and that double negations make explicit where classical reasoning—that is, interprocess communication—is used. These double negations also ensure that every $\widetilde{\rho}$ is itself a protocol type, a vital property if interprocess communication is to be flexible. It is not the case that $\widetilde{\widetilde{\rho}} = \rho$, but these types are isomorphic, as is demonstrated in Section 3.2.2.

The type level behavior of **go** is now clear: $\mathbf{go}^{\rho}$ witnesses the logical isomorphism between the intuitionistic negation of ρ—that is, $\rho \overset{\bullet}{\to} \bot$—and its doubly negated dual $\widetilde{\rho}$:

$$[\text{T-Go}]\ \frac{\Gamma; \Delta; \Pi \vdash e : \rho \overset{\bullet}{\to} \bot}{\Gamma; \Delta; \Pi \vdash \mathbf{go}^{\rho}\ e : \widetilde{\rho}}$$

Figure 3.3 gives the process evaluation rules for classical F$^{\circ}$. Rule EP-Go implements **go** as just described, but aside from bookkeeping of types it behaves exactly as seen previously. The next three rules show how **yield** allows synchronous communication between processes on channels at various types; in each case evaluation takes care of the outermost type constructor on the channel, reducing the type annotation on the ν binding—a

$$[\text{EP-GO}]\ \frac{a \text{ not free in } E\big[\mathbf{go}^{\rho}\ v\big]}{E\big[\mathbf{go}^{\rho}\ v\big] \longrightarrow \nu a{:}\rho.\ (E\big[\uparrow a\uparrow\big] \mid v \downarrow a\downarrow)}$$

$$[\text{EP-TRANSFER}]\ \nu a{:}\tau \overset{\bullet}{\to} \rho.\ E_1\big[\mathbf{yield}\ \uparrow a\uparrow\big] \mid E_2\big[\downarrow a\downarrow v\big] \longrightarrow \nu a{:}\rho.\ E_1\big[(v, \uparrow a\uparrow)\big] \mid E_2\big[\downarrow a\downarrow\big]$$

$$[\text{EP-BRANCH}]\ \nu a{:}\rho_1\ \&\ \rho_2.\ E_1\big[\mathbf{yield}\ \uparrow a\uparrow\big] \mid E_2\big[\downarrow a\downarrow.i\big] \longrightarrow \nu a{:}\rho_i.\ E_1\big[\mathbf{inj}_i^{\rho_1 \oplus \rho_2}\ \uparrow a\uparrow\big] \mid E_2\big[\downarrow a\downarrow\big]$$

$$[\text{EP-TTRANSFER}]\ \begin{array}{c} \nu a{:}\forall\alpha{:}\kappa.\ \rho.\ E_1\big[\mathbf{yield}\ \uparrow a\uparrow\big] \mid E_2\big[\downarrow a\downarrow\ [\tau]\big] \longrightarrow \\ \nu a{:}\rho.\ E_1\big[\mathbf{pack}\ \alpha{:}\kappa = \tau\ \mathbf{in}\ (\uparrow a\uparrow : \widetilde{\rho})\big] \mid E_2\big[\downarrow a\downarrow\big] \end{array}$$

$$[\text{EP-CLOSE}]\ \nu a{:}\perp.\ E_1\big[\mathbf{yield}\ \uparrow a\uparrow\big] \mid E_2\big[\downarrow a\downarrow\big] \longrightarrow E_1\big[()\big] \mid \nu a{:}\perp.\ E_2\big[\rfloor a\lfloor\big]$$

$$[\text{EP-DONE}]\ P \mid \nu a{:}\perp.\ \rfloor a\lfloor \longrightarrow P$$

$$[\text{EP-PAR}]\ \frac{P_1 \longrightarrow P_1'}{P_1 \mid P_2 \longrightarrow P_1' \mid P_2} \qquad [\text{EP-NEW}]\ \frac{P \longrightarrow P'}{\nu a{:}\tau.\ P \longrightarrow \nu a{:}\tau.\ P'}$$

Figure 3.3: Process evaluation rules for classical F°

bit of bookkeeping required for soundness, as the elimination form used at a sink endpoint uniquely identifies a single process evaluation rule that applies.

Rule EP-TRANSFER sends a value across a channel as shown previously, but this time the **yield** produces a pair (as $\otimes$ is the dual of $\to$) of the transferred value and the source endpoint itself, now at different type—*i.e.*, the next step of the protocol. EP-BRANCH relays the projection taken from the sink endpoint by means of an injection of the source endpoint (as $\oplus$ is the dual of $\&$), again adjusting the channel's type annotation accordingly. EP-TTRANSFER sends a type across a channel exactly as EP-TRANSFER sends a value, and likewise the source endpoint appears within an existential package (as $\exists$ is the dual of $\forall$).

Rule EP-CLOSE similarly involves interaction between two processes, but the interaction in this case is trivial: the **yield** does not need to block until the other process is in any particular state, the channel type annotation does not change, and no information is communicated; **yield** simply returns $()$, while the sink endpoint $\downarrow a\downarrow$ is replaced by the closed channel $\rfloor a\lfloor$, both of which are of type $\perp$. EP-DONE allows a process consisting of a closed channel and its ν binder to vanish, while EP-PAR and EP-NEW allow evaluation to take place within a non-atomic process—it is sufficient, in EP-PAR, to allow evaluation only on

$$[\text{EQP-REFL}]\ P \equiv P \qquad [\text{EQP-SYM}]\ \frac{P_2 \equiv P_1}{P_1 \equiv P_2} \qquad [\text{EQP-TRANS}]\ \frac{P_1 \equiv P_2 \qquad P_2 \equiv P_3}{P_1 \equiv P_3}$$

$$[\text{EQP-COMM}]\ P_1 \mid P_2 \equiv P_2 \mid P_1 \qquad [\text{EQP-PAR}]\ \frac{P_1 \equiv P_1' \qquad P_2 \equiv P_2'}{P_1 \mid P_2 \equiv P_1' \mid P_2'}$$

$$[\text{EQP-NEW}]\ \frac{P \equiv P'}{\nu a{:}\rho.\ P \equiv \nu a{:}\rho.\ P'} \qquad [\text{EQP-ASSOC}]\ (P_1 \mid P_2) \mid P_3 \equiv P_1 \mid (P_2 \mid P_3)$$

$$[\text{EQP-SWAP}]\ \nu a_1{:}\rho_1.\ \nu a_2{:}\rho_2.\ P \equiv \nu a_2{:}\rho_2.\ \nu a_1{:}\rho_1.\ P$$

$$[\text{EQP-EXTRUDE}]\ \frac{a \text{ not free in } P_2}{(\nu a{:}\rho.\ P_1) \mid P_2 \equiv \nu a{:}\rho.\ (P_1 \mid P_2)}$$

Figure 3.4: Process equivalence rules for classical F°

the left side of a parallel composition, as, like in many π-calculus-inspired systems, process evaluation in classical F° is always considered with respect to an equivalence relation $\equiv$ on processes, given in Figure 3.4. The definition of $\equiv$ is completely standard, establishing general properties of equivalence relations, commutativity and associativity of parallel composition, and scope extrusion and reordering for the binding of channel names.

Two final points must be addressed by operational semantics: because $\uparrow\tau\uparrow$ is simply $(\tau \overset{\bullet}{\to} \bot) \overset{\bullet}{\to} \bot$, it is possible to apply source endpoints and to **yield** on expressions other than those that evaluate to source endpoints. Figure 3.5 gives appropriate congruence rules for these situations. For the case of a source in the function position of an application, recall that other term languages for classical logics (Girard, 1987; Parigot, 1992) insist that the interaction between terms of a type and its negation not depend on the ordering of those terms. Thus, applying $\uparrow a\uparrow$ of type $(\tau \overset{\bullet}{\to} \bot) \overset{\bullet}{\to} \bot$ to v of type $\tau \overset{\bullet}{\to} \bot$ should be equivalent to **yield**ing on $\uparrow a\uparrow$ and supplying the result to v. Rule E-SOURCE makes this so.

The case of **yield**ing on other sorts of values is reminiscent of the briefly considered linear variant of **control**, which combined the functionality of **go** and **yield**. The rules E-LAMBDA and EP-SINK thus synthesize a **go** in these cases, spawning a new process with a simple protocol and waiting for its reply. These rules must also create **let** expressions around the modified **yield** expressions, as, unlike in the presentation that followed **con-**

$$[\text{E-SOURCE}]\ \uparrow a \uparrow\ v \longrightarrow v\ (\mathbf{yield}\ \uparrow a \uparrow)$$

$$[\text{E-LAMBDA}]\ \frac{u, z \notin \text{fv}(e)}{\mathbf{yield}\ \lambda^{\bullet} x{:}\tau \xrightarrow{\bullet} \bot.\ e \longrightarrow \mathbf{let}\ (z, u) = \mathbf{yield}\ (\mathbf{go}^{\tau \xrightarrow{\bullet} \bot}\ \lambda^{\bullet} x{:}\tau \xrightarrow{\bullet} \bot.\ e)\ \mathbf{in}\ \mathbf{yield}\ u; z}$$

$$[\text{EP-SINK}]\ \nu a{:}\uparrow\tau\uparrow.\ E\big[\mathbf{yield}\ \downarrow a \downarrow\big] \longrightarrow \nu a{:}\uparrow\tau\uparrow.\ E\big[\mathbf{let}\ (z, u) = \mathbf{yield}\ (\mathbf{go}^{\tau \xrightarrow{\bullet} \bot}\ \downarrow a \downarrow)\ \mathbf{in}\ \mathbf{yield}\ u; z\big]$$

Figure 3.5: Congruence rules for classical F°

trol, it is not actually the case that $\mathbf{yield}\ (\mathbf{go}^{\tau \xrightarrow{\bullet} \bot}\ \ldots)$ will simply return a value of type τ; dualization instead leads to the isomorphic $\tau \otimes \uparrow\mathbf{1}\uparrow$.

Though these congruence rules may appear unintuitive in isolation, they serve to complete the picture wherein standard logical connectives serve double duty as constructors for session types, and the behavior they specify is exactly what is needed in these corner cases. The next section will argue, through a series of examples, that this is indeed a compelling picture, demonstrating how the modest machinery introduced thus far can scale up.

3.2 Examples

Concurrency as presented in classical F° so far has been very simple, conveying simply one value, type, or bit of branching information per message in an apparently unidirectional manner. This section demonstrates that bigger constructs can be built from these simple pieces, starting with one of the simplest well-known concurrency primitives and ending with an idealized Diffie-Hellman handshake protocol.

3.2.1 Futures

A *future* (Niehren et al., 2006) is simply a sub-computation to be calculated in a separate thread; the main computation will wait for this thread to complete when the future's value

is needed. This simple form of concurrency is easy to express in classical F$^\circ$:

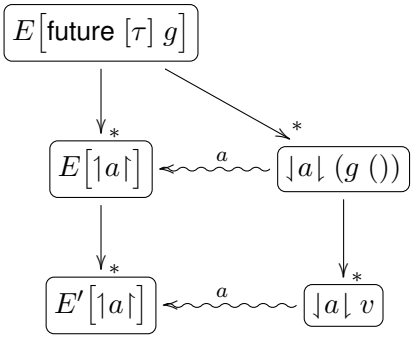
$$\mathsf{Future}[\tau] \triangleq \lceil \tau \otimes \mathbf{1} \rceil$$

$$\begin{aligned} \mathsf{future} &: \forall\alpha{:}\circ.\ (\mathbf{1} \xrightarrow{\circ} \alpha) \xrightarrow{\star} \mathsf{Future}[\alpha] \\ \mathsf{future} &\triangleq \Lambda\alpha{:}\circ.\ \lambda^\circ x{:}\mathbf{1} \xrightarrow{\circ} \alpha.\ \mathbf{go}^{\alpha \xrightarrow{\bullet} \perp}\ \lambda^\circ k{:}\alpha \xrightarrow{\bullet} \perp.\ k\ (x\ ()) \end{aligned}$$

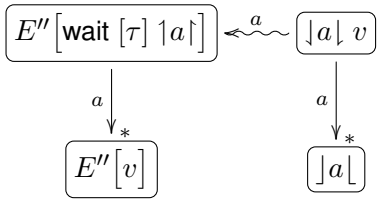
$$\begin{aligned} \mathsf{wait} &: \forall\alpha{:}\circ.\ (\mathsf{Future}[\alpha]) \xrightarrow{\star} \alpha \\ \mathsf{wait} &\triangleq \Lambda\alpha{:}\circ.\ \lambda^\circ f{:}\mathsf{Future}[\alpha].\ \mathbf{let}\ (z, u) = \mathbf{yield}\ f\ \mathbf{in}\ \mathbf{yield}\ u; z \end{aligned}$$

The main process passes a thunk to its newly spawned child; this child applies the thunk and sends back the result.

Visually, the run-time behavior of $E\big[\mathsf{future}\ [\tau]\ g\big]$, where $g\ ()\ \longrightarrow^*\ v$ and $E\big[\,-\,\big]\ \longrightarrow^*\ E'\big[\,-\,\big]$, is



The connection between endpoints of a channel at a given moment in time is given by the $\rightsquigarrow$ arrow. Similarly, for such some $\lceil a \rceil$ of type $\mathsf{Future}[\tau]$, we have



Here the a label on evaluation arrows indicates that communication over a has occurred. Since a supports no further communication afterwards—its sink endpoint has been replaced by the closed channel $\lfloor a \rfloor$—the $\rightsquigarrow$ connection is then removed. Recall that such a lone $\lfloor a \rfloor$ indicates a completed process; the child process in this example is now finished and can disappear.

3.2.2 Linking channel endpoints

Given v_{src} of type $\uparrow\tau\uparrow$ and v_{snk} of type $\tau \overset{\circ}{\to} \bot$—which may or may not be literal source and sink endpoints—one might wish to join the two such that v_{src} flows to v_{snk} without making the parent process **yield** on v_{src}. Doing so, however, does not free the parent from the need to return a value of type $\bot$—this cannot be the value that applying v_{snk} would produce, so it must come from some other process.

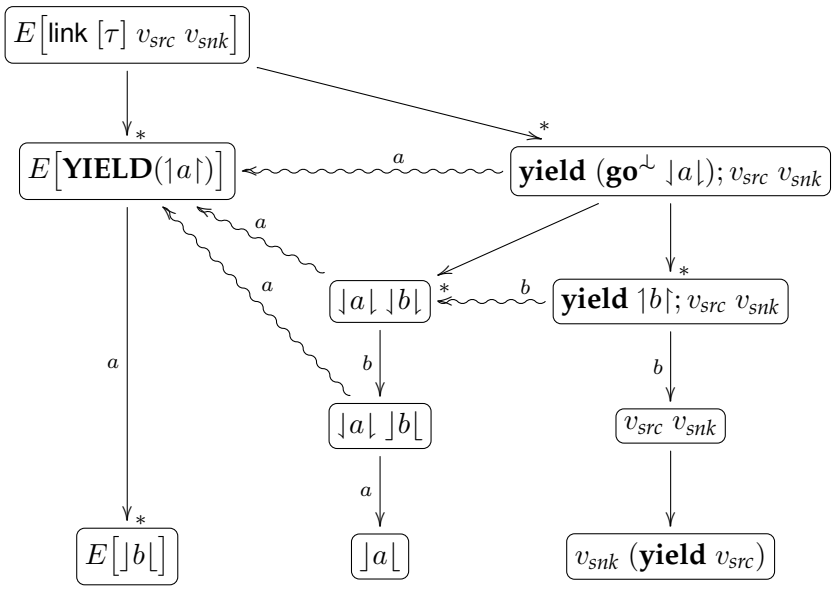
The solution relies on the fact that closed channels are first class objects and can thus be passed from one process to another:

$$\begin{aligned}
\mathsf{link} \quad &: \quad \forall\alpha{:}\circ.\ \uparrow\alpha\uparrow \overset{\star}{\to} (\alpha \overset{\bullet}{\to} \bot) \overset{\circ}{\to} \bot \\
\mathsf{link} \quad &\triangleq \quad \Lambda\alpha{:}\circ.\ \lambda^{\star}x{:}\uparrow\alpha\uparrow.\ \lambda^{\circ}f{:}\alpha \overset{\bullet}{\to} \bot.\ \mathbf{yield}\ \lambda^{\bullet}g{:}\bot \overset{\bullet}{\to} \bot.\ \mathbf{yield}\ g; x\ f
\end{aligned}$$

Note that the final $x\ f$ will step to $f\ (\mathbf{yield}\ x)$ via E-SOURCE; similarly, E-LAMBDA and E-SOURCE will insert $\mathbf{go}^{\bot \overset{\bullet}{\to} \bot}$ and $\mathbf{go}^{\bot}$ immediately following the **yield**s that are already present. A call to $\mathsf{link}\ [\tau]\ v_{src}\ v_{snk}$ thus spawns two processes: the first spawns the second with the trivial protocol $\bot$ and the first child's own sink endpoint, $\lfloor a \rfloor$, as its body; then it proceeds to wait and link the original arguments. The second applies its body ($\lfloor a \rfloor$) to its own trivial sink endpoint ($\lfloor b \rfloor$) of type $\bot$—which may step immediately to the closed channel $\rfloor b \lfloor$—thus uses $\lfloor a \rfloor$ to pass either $\lfloor b \rfloor$ or $\rfloor b \lfloor$ (depending on the order in which parallel evaluations are resolved) back to the parent process. This is illustrated in Figure 3.6 for the case where the innermost **yield** resolves before the outermost—the other case is functionally equivalent. The notation $\mathbf{YIELD}(e)$ abbreviates the now common pattern of **yield**ing to receive a product with type $\uparrow\mathbf{1}\uparrow$ on the right, immediately unpacking the resulting pair, and eliminating the right component via another **yield** and sequencing.

Bidirectional communication So far we have seen child processes send information back to their parents. While the constructs of classical F$^{\circ}$ do immediately suggest this sort of communication, exchanges in both directions are quite possible; the unidirectional appearance of dualization simply requires functions that explicitly witness the isomorphism from $\widetilde{\widetilde{\rho}}$ to ρ.

For example, while the dual of $\tau \overset{\bullet}{\to} \rho$ is $\uparrow\tau \otimes \widetilde{\rho}\uparrow$, the dual of $\uparrow\tau \otimes \rho\uparrow$ is the somewhat unwieldy $\uparrow((\tau \otimes \rho) \overset{\bullet}{\to} \bot) \otimes \uparrow\mathbf{1}\uparrow\uparrow$ rather than the $\tau \overset{\bullet}{\to} \widetilde{\rho}$ for which one might have hoped.



$$\mathbf{YIELD}(e) \triangleq \mathbf{let}\ (z,u) = \mathbf{yield}\ e\ \mathbf{in}\ \mathbf{yield}\ u; z$$

Figure 3.6: Evaluation of $\mathsf{link}\ [\tau]\ v_{src}\ v_{snk}$

Yet the former can be transformed into the latter with a **yield** operation, an uncurrying, a partial application, and a **go**; these steps can be combined into a function send:

$$\begin{aligned}
\mathsf{send} \quad &: \quad \forall\alpha{:}\circ.\ \forall\beta{:}\bullet.\ \widetilde{\uparrow\alpha \otimes \beta\uparrow} \xrightarrow{\star} \alpha \xrightarrow{\bullet} \widetilde{\beta} \\
\mathsf{send} \quad &\triangleq \quad \Lambda\alpha{:}\circ.\ \Lambda\beta{:}\bullet.\ \lambda^{\star}s{:}\widetilde{\uparrow\alpha \otimes \beta\uparrow}. \\
&\qquad\qquad \mathbf{let}\ (f,u) = \mathbf{yield}\ s\ \mathbf{in} \\
&\qquad\qquad\quad \mathbf{yield}\ u; \lambda^{\bullet}x{:}\alpha.\ \mathbf{go}^{\beta}\ \lambda^{\bullet}p{:}\beta.\ f\ (x,p)
\end{aligned}$$

Similarly, the dual of $\uparrow\rho_1 \oplus \rho_2\uparrow$ is $\uparrow((\rho_1 \oplus \rho_2) \xrightarrow{\bullet} \perp) \otimes \uparrow\mathbf{1}\uparrow\uparrow$; to coerce it to $\widetilde{\rho_1} \,\&\, \widetilde{\rho_2}$, define select:

$$\begin{aligned}
\mathsf{select} \quad &: \quad \forall\alpha{:}\bullet.\ \forall\beta{:}\bullet.\ \widetilde{\uparrow\alpha \oplus \beta\uparrow} \xrightarrow{\star} \widetilde{\alpha} \,\&\, \widetilde{\beta} \\
\mathsf{select} \quad &\triangleq \quad \Lambda\alpha{:}\bullet.\ \Lambda\beta{:}\bullet.\ \lambda^{\star}s{:}\widetilde{\uparrow\alpha \oplus \beta\uparrow}. \\
&\qquad\qquad \mathbf{let}\ (f,u) = \mathbf{yield}\ s\ \mathbf{in} \\
&\qquad\qquad\quad \mathbf{yield}\ u; \langle \mathbf{go}^{\alpha}\ \lambda^{\bullet}p_1{:}\alpha.\ f\ \mathbf{inj}_1^{\alpha\oplus\beta}\ p_1, \mathbf{go}^{\beta}\ \lambda^{\bullet}p_2{:}\beta.\ f\ \mathbf{inj}_2^{\alpha\oplus\beta}\ p_2\rangle
\end{aligned}$$

Unfortunately, the dependency present in universal and existential types makes it impossible to write a general purpose coercion send_type from $\widetilde{\uparrow\exists\alpha{:}\kappa.\ \beta\uparrow}$ to $\forall\alpha{:}\kappa.\ \widetilde{\beta}$—those types,

as written, miss the fact that any interesting type substituted for β will include occurrences of α. Directionality can still be reversed on a case-by-case basis, however, and Section 4.5 discusses what additional quantification is necessary to implement send_type.

To demonstrate send, let us examine the identity function echo, which spawns a child process, passes its argument to that child, then receives that argument back:

$$\begin{aligned}
\mathsf{reply} &: \forall\alpha{:}\circ.\, \uparrow\alpha \otimes (\alpha \xrightarrow{\bullet} \downarrow)\uparrow \xrightarrow{\bullet} \downarrow \\
\mathsf{reply} &= \Lambda\alpha{:}\circ.\, \lambda^{\bullet}h{:}\uparrow\alpha \otimes (\alpha \xrightarrow{\bullet} \downarrow)\uparrow.\, \mathbf{let}\ (y, g) = \mathbf{yield}\ h\ \mathbf{in}\ g\ y \\
\\
\mathsf{echo} &: \forall\alpha{:}\circ.\, \alpha \xrightarrow{\star} \alpha \\
\mathsf{echo} &= \Lambda\alpha{:}\circ.\, \lambda^{\star}x{:}\alpha.\, \mathbf{let}\ (z, u) = \mathbf{yield} \\
&\qquad\qquad \mathsf{send}\ [\alpha]\ [\alpha \xrightarrow{\bullet} \downarrow]\ (\mathbf{go}^{\uparrow\alpha\otimes(\alpha\xrightarrow{\bullet}\downarrow)\uparrow}\ \mathsf{reply}\ [\alpha])\ x \\
&\qquad\qquad \mathbf{in\ yield}\ u; z
\end{aligned}$$

Here reply is the body of the child process that will receive echo's argument and send it back. (The type of reply could have been written as the equivalent $\forall\alpha{:}\circ.\, \uparrow\alpha \otimes (\alpha \xrightarrow{\bullet} \downarrow) \xrightarrow{\bullet} \downarrow\uparrow$; this notation of the type would better reflects how it is used within echo, whereas the notation given above more closely matches its definition.)

The execution of echo $[\tau]\ v$ for some v of type τ is shown in Figure 3.7. We can see how, while the initial spawning of the reply process orients the channel a in the usual child-to-parent direction, the machinery of send spawns another process that sets up a channel b in the opposite direction; afterwards, a third channel c is established in the original direction. All this is facilitated quite naturally by classical F°'s congruence rules.

It is worth noting that, while v moves among several processes, at no point does a cycle exist in the communication structure—the $\rightsquigarrow$ arrows—of Figure 3.7. That such acyclicity always holds is crucial to the proof of soundness in Section 3.3.1.

3.2.3 Diffie-Hellman key exchange

So far we have seen relatively small examples. As a larger demonstration of the protocols expressible in classical F°, let us consider Diffie-Hellman key exchange, which is defined as follows:

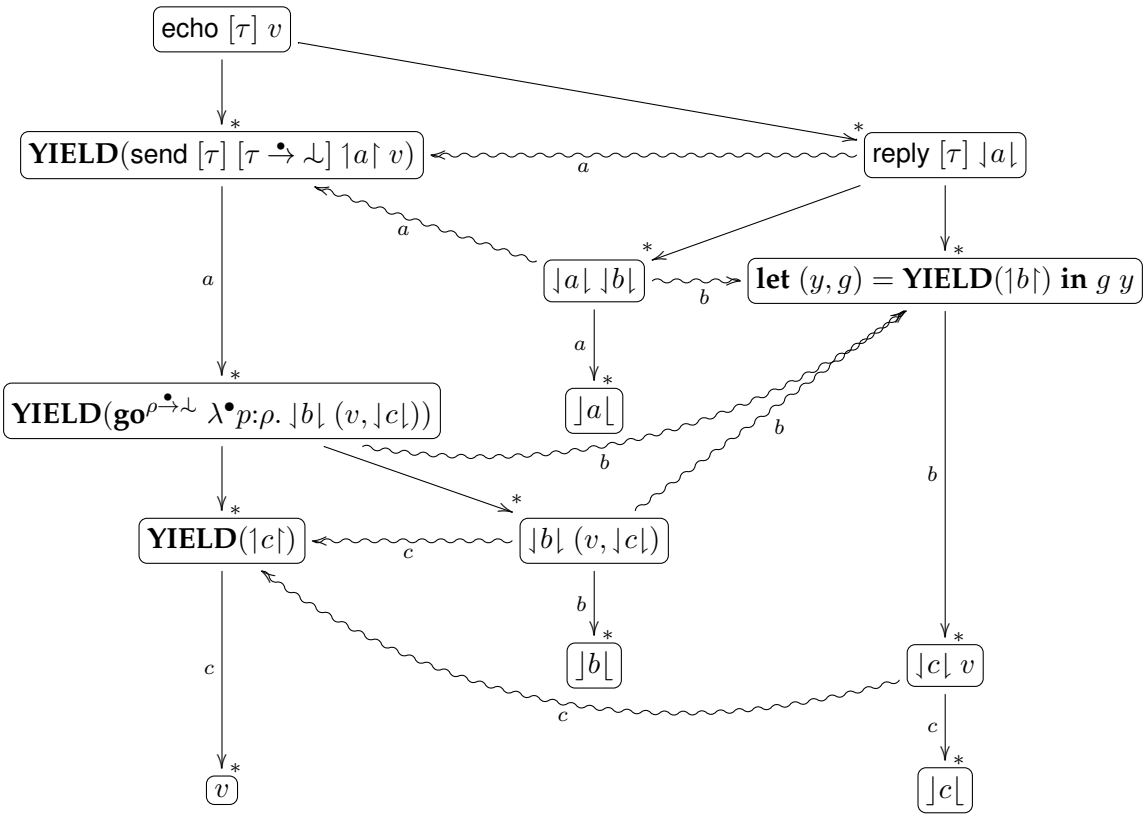


Figure 3.7: Evaluation of echo $[\tau]$ v

1. Alice and Bob select secret integers a and b.
2. Alice and Bob exchange g^a mod p and g^b mod p in the clear.
3. Alice and Bob compute the shared secret $(g^b)^a = (g^a)^b$ mod p and use it to encrypt further communication.

Here g is a publicly known generator with certain properties, often 2 or 5, and p is a similarly known large prime number. The shared secret $(g^b)^a = (g^a)^b$ mod p cannot feasibly be computed from the publicly known values g^a and g^b.

For purposes of this example, let us augment F$^\circ$ with the types Int and String of kind $\star$, along with the following operations, which behave as suggested by their names:

$$\begin{aligned}
\mathsf{big_random} &: \mathbf{1} \xrightarrow{\star} \mathsf{Int} \\
\mathsf{pow_mod} &: \mathsf{Int} \xrightarrow{\star} \mathsf{Int} \xrightarrow{\star} \mathsf{Int} \xrightarrow{\star} \mathsf{Int} \\
\mathsf{less_than} &: \mathsf{Int} \xrightarrow{\star} \mathsf{Int} \xrightarrow{\star} \mathbf{1} \oplus \mathbf{1} \\
\mathsf{encrypt} &: \forall\alpha{:}\star.\ \mathsf{Int} \xrightarrow{\star} \alpha \xrightarrow{\star} \mathsf{String} \\
\mathsf{decrypt} &: \forall\alpha{:}\star.\ \mathsf{Int} \xrightarrow{\star} \mathsf{String} \xrightarrow{\star} \mathbf{1} \oplus \alpha
\end{aligned}$$

As was done in Section 2.2, this example uses standard single variable **let** expressions in addition to those that eliminate multiplicative products.

There is, of course, no network over which Alice and Bob can communicate in F$^\circ$, nor is there a mechanism by which two separate expressions might later learn of each other, so let us define Alice as a child process spawned by Bob for purposes of this demonstration. If either party is allowed to quit the handshake at any time (a very reasonable allowance), and if the post-handshake portion of the exchange is represented by α, then the protocol from Alice's perspective is

$$\mathsf{Alice}[\alpha] \quad = \quad \mathsf{Int} \xrightarrow{\bullet} \uparrow\downarrow \oplus \uparrow\mathsf{Int} \otimes (\downarrow \,\&\, \alpha)\uparrow\uparrow$$

Reading this type reveals exactly how Alice participates in the handshake: she sends an Int (her public key), then waits to see whether Bob decided to end the exchange or to continue; if the latter, Alice waits for an Int (Bob's public key) and then may decide whether to abort or continue as α.

The function dh_alice, which serves as the body for the child process that acts as Alice, takes as arguments the post-handshake protocol type α, the Diffie-Hellman parameters g and p, a threshold n which g^b mod p must be no smaller than (a questionable requirement in practice but one that allows for simple "abort" branches), a continuation c to handle the

post-handshake part of the exchange, and an initial channel of type Alice.

$$
\begin{aligned}
\mathsf{dh\_alice} \quad &: \quad \forall\alpha{:}\bullet.\ \mathsf{Int} \xrightarrow{\star} \mathsf{Int} \xrightarrow{\star} \mathsf{Int} \xrightarrow{\star} (\mathsf{Int} \xrightarrow{\star} \alpha \xrightarrow{\bullet} \bot) \xrightarrow{\bullet} \mathsf{Alice} \xrightarrow{\bullet} \bot \\
\mathsf{dh\_alice} \quad &= \quad \Lambda\alpha{:}\bullet.\ \lambda^{\star} g{:}\mathsf{Int}.\ \lambda^{\star} p{:}\mathsf{Int}.\ \lambda^{\star} n{:}\mathsf{Int}.\ \lambda^{\bullet} c{:}\mathsf{Int} \xrightarrow{\star} \alpha \xrightarrow{\bullet} \bot.\ \lambda^{\bullet} s{:}\mathsf{Alice}. \\
&\qquad \mathbf{let}\ a = \mathsf{big\_random}\ ()\ \mathbf{in} \\
&\qquad \mathbf{case}\ \mathbf{yield}\ (s\ (\mathsf{pow\_mod}\ g\ a\ p))\ \mathbf{of} \\
&\qquad\quad \mathbf{inj}_1\ s_1 \mapsto s_1 \\
&\qquad\quad \mid \mathbf{inj}_2\ s_2 \mapsto \mathbf{let}\ (b, s_3) = \mathbf{yield}\ s_2\ \mathbf{in} \\
&\qquad\quad\ \ \mathbf{case}\ \mathsf{less\_than}\ b\ n\ \mathbf{of} \\
&\qquad\quad\ \ \ \mathbf{inj}_1\ u_1 \mapsto u_1; s_3.1 \\
&\qquad\quad\ \ \mid \mathbf{inj}_2\ u_2 \mapsto u_2; c\ (\mathsf{pow\_mod}\ b\ a\ p)\ s_3.2
\end{aligned}
$$

Since Alice's session is the child process, she follows through with an abort by simply returning the completed session (of type $\bot$) at the appropriate point, whether the decision to abort was made by her or by Bob.

An implementation of Bob's side of the exchange—*i.e.*, the parent process—looks very similar. Because Bob is the parent, dh_bob must take its own continuation (c_b) as well as Alice's (c_a); it immediately launches dh_alice with c_a as its own process, then proceeds with its side of the Diffie-Hellman handshake. An additional type argument β is also provided, representing the final type of the exchange should it be successful—in cases where the handshake fails, an error message is returned instead. (At times dh_bob must make use of the select and send functions introduced earlier. Their type arguments are always the left and right components of the portion of Alice that has yet to be consumed—*i.e.*, the remainder of the protocol.)

$$
\begin{array}{lcl}
\mathsf{dh\_bob} & : & \forall\alpha{:}\bullet.\ \forall\beta{:}\circ.\ \mathsf{Int} \xrightarrow{\star} \mathsf{Int} \xrightarrow{\star} \mathsf{Int} \xrightarrow{\star} (\mathsf{Int} \xrightarrow{\star} \alpha \xrightarrow{\bullet} \bot) \xrightarrow{\circ} (\mathsf{Int} \xrightarrow{\star} \widetilde{\alpha} \xrightarrow{\circ} \beta) \xrightarrow{\circ} \mathsf{String} \oplus \beta \\
\mathsf{dh\_bob} & = & \Lambda\alpha{:}\bullet.\ \Lambda\beta{:}\circ.\ \lambda^{\star}g{:}\mathsf{Int}.\ \lambda^{\star}p{:}\mathsf{Int}.\ \lambda^{\star}n{:}\mathsf{Int}.\ \lambda^{\circ}c_A{:}\mathsf{Int} \xrightarrow{\star} \alpha \xrightarrow{\bullet} \bot.\ \lambda^{\circ}c_B{:}\mathsf{Int} \xrightarrow{\star} \widetilde{\alpha} \xrightarrow{\circ} \beta. \\
& & \quad \mathbf{let}\ (a, s) = \mathbf{yield}\ (\mathbf{go}^{\mathsf{Alice}[\alpha]}\ (\mathsf{dh\_alice}\ [\alpha]\ g\ p\ n\ c_A))\ \mathbf{in} \\
& & \quad \mathbf{let}\ s_0 = \mathsf{select}\ [\bot]\ [\uparrow \mathsf{Int} \otimes (\bot\ \&\ \alpha)\uparrow]\ s\ \mathbf{in} \\
& & \quad \mathbf{case}\ \mathsf{less\_than}\ a\ n\ \mathbf{of} \\
& & \qquad \mathbf{inj}_1\ u_1 \mapsto u_1; s_0.1; \mathbf{inj}_1^{\mathsf{String}\oplus\beta}\ \texttt{"Alice's public key rejected."} \\
& & \quad \mid \mathbf{inj}_2\ u_2 \mapsto u_2; \mathbf{let}\ s_1 = s_0.2\ \mathbf{in} \\
& & \qquad \mathbf{let}\ b = \mathsf{big\_random}\ b\ \mathbf{in} \\
& & \qquad \mathbf{let}\ s_2 = \mathsf{send}\ [\mathsf{Int}]\ [\bot\ \&\ \alpha]\ s_1\ (\mathsf{pow\_mod}\ g\ b\ p)\ \mathbf{in} \\
& & \qquad \mathbf{case}\ \mathbf{yield}\ s_2\ \mathbf{of} \\
& & \qquad\quad \mathbf{inj}_1\ u \mapsto \mathbf{yield}\ u; \mathbf{inj}_1^{\mathsf{String}\oplus\beta}\ \texttt{"Bob's public key rejected."} \\
& & \qquad \mid \mathbf{inj}_2\ s_3 \mapsto \mathbf{inj}_2^{\mathsf{String}\oplus\beta}\ (c_B\ (\mathsf{pow\_mod}\ a\ b\ p)\ s_3)
\end{array}
$$

The continuation functions representing the bodies of Alice and Bob's respective sides of the exchange can be as simple or as complex as necessary. If Alice is simply going to send an encrypted integer and quit—in which case the remainder of the protocol from Alice's perspective (α) is $\mathsf{String} \xrightarrow{\bullet} \bot$, as every encrypted value is a string—then this can be

$$
\begin{array}{lcl}
\mathsf{answer} & : & \mathsf{Int} \xrightarrow{\star} (\mathsf{String} \xrightarrow{\bullet} \bot) \xrightarrow{\bullet} \bot \\
\mathsf{answer} & = & \lambda^{\star}k{:}\mathsf{Int}.\ \lambda^{\bullet}s{:}\mathsf{String} \xrightarrow{\bullet} \bot.\ s\ (\mathsf{encrypt}\ [\mathsf{Int}]\ k\ 42)
\end{array}
$$

Any corresponding continuation for Bob must follow the opposite side of the communication protocol—*i.e.*, $\uparrow\mathsf{String} \otimes \mathbf{1}\uparrow$—but, apart from this constraint, it can behave however it likes. If $\mathbf{1} \oplus \mathsf{Int}$ is supplied for β, Bob can attempt to decrypt Alice's message and simply return it if possible with

$$
\begin{array}{lcl}
\mathsf{broadcast} & : & \mathsf{Int} \xrightarrow{\star} \uparrow\mathsf{String} \otimes \mathbf{1}\uparrow \xrightarrow{\circ} \mathbf{1} \oplus \mathsf{Int} \\
\mathsf{broadcast} & = & \lambda^{\star}k{:}\mathsf{Int}.\ \lambda^{\circ}s{:}\uparrow\mathsf{String} \otimes \mathbf{1}\uparrow.\ \mathbf{let}\ (z, u) = \mathbf{yield}\ s\ \mathbf{in}\ \mathbf{yield}\ u; \mathsf{decrypt}\ [\mathsf{Int}]\ z
\end{array}
$$

While illustrating its evaluation along the lines of the previous examples would be tedious, there would be no surprises involved in showing that, with very high probability for any sufficiently large prime number p,

$$\mathsf{dh\_bob}\ [\mathsf{String} \overset{\bullet}{\to} \lightning]\ [\mathbf{1} \oplus \mathsf{Int}]\ 2\ p\ 4294967296\ \mathsf{answer\ broadcast} \longrightarrow^* \mathbf{inj}_2\ (\mathbf{inj}_2\ 42)$$

More complex continuations for Alice and Bob would likely use send and select for bidirectional communication, and, since the possibility of a decryption error exists at every step, both sides must be on guard for early termination—the type system will stop either party from forgetting that the other may have quit.

3.3 Metatheory

Having demonstrated the applicability of classical F°, it is now time to verify its soundness. As in Section 2.3, this will include both a traditional soundness proof and arguments based on semantics augmented with annotations.

The expression typing rules for classical F° not already presented in Section 3.1 can be seen in Figure 3.9. Aside from the addition of the channel context Π, most of these rules are exactly as they appeared in Figure 2.5, although T-LAM now includes the same condition present in K-ARROW, requiring that the result type of a function have kind $\bullet$ if the function type itself has that kind. Figure 3.8 updates nondeterministic context extension for classical F° by adding Π; note that B-LIN suffices for variables of kind $\bullet$, so no additional rule is needed.

Rules T-SINK, T-SOURCE, and T-CLOSED type channel endpoints and closed channels according to the channel context Π—like Δ, Π is linear and must contain exactly one bind-

$$[\text{B-LIN}]\ \frac{\Gamma \vdash \tau : \circ \qquad \Gamma; \Delta, x{:}\tau; \Pi \vdash e{:}\tau' \qquad x \notin \Gamma, \Delta}{[\Gamma; \Delta], x{:}\tau; \Pi \vdash e{:}\tau'}$$

$$[\text{B-UN}]\ \frac{\Gamma \vdash \tau : \star \qquad \Gamma, x{:}\tau; \Delta; \Pi \vdash e{:}\tau' \qquad x \notin \Gamma, \Delta}{[\Gamma; \Delta], x{:}; \Pi\tau \vdash e{:}\tau'}$$

Figure 3.8: Auxiliary judgments for classical F° expression typing

$$[\text{T-LVAR}]\ \Gamma; x{:}\tau; \cdot \vdash x : \tau \qquad [\text{T-UVAR}]\ \frac{x{:}\tau \in \Gamma}{\Gamma; \cdot; \cdot \vdash x : \tau}$$

$$[\text{T-LAM}]\ \frac{[\Gamma; \Delta], x{:}\tau_1; \Pi \vdash e : \tau_2 \qquad \kappa = \star \implies \Delta = \cdot \qquad \kappa = \bullet \implies \Gamma \vdash \tau_2 : \bullet}{\Gamma; \Delta; \Pi \vdash \lambda^{\kappa} x{:}\tau_1.\ e : \tau_1 \xrightarrow{\kappa} \tau_2}$$

$$[\text{T-APP}]\ \frac{\Gamma; \Delta_1; \Pi_1 \vdash e_1 : \tau_1 \xrightarrow{\kappa} \tau_2 \qquad \Gamma; \Delta_2; \Pi_2 \vdash e_2 : \tau_1}{\Gamma; \Delta_1 \Cup \Delta_2; \Pi_1 \Cup \Pi_2 \vdash e_1\ e_2 : \tau_2}$$

$$[\text{T-APAIR}]\ \frac{\Gamma; \Delta; \Pi \vdash e_1 : \tau_1 \qquad \Gamma; \Delta; \Pi \vdash e_2 : \tau_2}{\Gamma; \Delta; \Pi \vdash \langle e_1, e_2 \rangle : \tau_1 \,\&\, \tau_2} \qquad [\text{T-SELECT}]\ \frac{\Gamma; \Delta; \Pi \vdash e : \tau_1 \,\&\, \tau_2}{\Gamma; \Delta; \Pi \vdash e.i : \tau_i}$$

$$[\text{T-TLAM}]\ \frac{\Gamma, \alpha{:}\kappa; \Delta; \Pi \vdash v : \tau \qquad \alpha \notin \Gamma}{\Gamma; \Delta; \Pi \vdash \Lambda\alpha{:}\kappa.\ v : \forall\alpha{:}\kappa.\ \tau} \qquad [\text{T-TAPP}]\ \frac{\Gamma; \Delta; \Pi \vdash e : \forall\alpha{:}\kappa.\ \tau' \qquad \Gamma \vdash \tau : \kappa}{\Gamma; \Delta; \Pi \vdash e\ [\tau] : \{\alpha \mapsto \tau\}\tau'}$$

$$[\text{T-UNIT}]\ \Gamma; \cdot; \cdot \vdash () : \mathbf{1} \qquad [\text{T-SEQ}]\ \frac{\Gamma; \Delta_1; \Pi_1 \vdash e_1 : \mathbf{1} \qquad \Gamma; \Delta_2; \Pi_2 \vdash e_2 : \tau}{\Gamma; \Delta_1 \Cup \Delta_2; \Pi_1 \Cup \Pi_2 \vdash e_1; e_2 : \tau}$$

$$[\text{T-MPAIR}]\ \frac{\Gamma; \Delta_1; \Pi_1 \vdash e_1 : \tau_1 \qquad \Gamma; \Delta_2; \Pi_2 \vdash e_2 : \tau_2}{\Gamma; \Delta_1 \Cup \Delta_2; \Pi_1 \Cup \Pi_2 \vdash (e_1, e_2) : \tau_1 \otimes \tau_2}$$

$$[\text{T-LET}]\ \frac{\Gamma; \Delta_1; \Pi_1 \vdash e_1 : \tau_1 \otimes \tau_2 \qquad [\Gamma; \Delta_2], x_1{:}\tau_1, x_2{:}\tau_2; \Pi_2 \vdash e_2 : \tau}{\Gamma; \Delta_1 \Cup \Delta_2; \Pi_1 \Cup \Pi_2 \vdash \mathbf{let}\ (x_1, x_2) = e_1\ \mathbf{in}\ e_2 : \tau}$$

$$[\text{T-IN}]\ \frac{\Gamma; \Delta; \Pi \vdash e : \tau_i \qquad \Gamma \vdash \tau_1 : \kappa_1 \qquad \Gamma \vdash \tau_2 : \kappa_2}{\Gamma; \Delta; \Pi \vdash \mathbf{inj}_i^{\tau_1 \oplus \tau_2}\ e : \tau_1 \oplus \tau_2}$$

$$[\text{T-CASE}]\ \frac{\Gamma; \Delta_1; \Pi_1 \vdash e : \tau_1 \oplus \tau_2 \qquad [\Gamma; \Delta_2], x_1{:}\tau_1; \Pi_2 \vdash e_1{:}\tau \qquad [\Gamma; \Delta_2], x_2{:}\tau_2; \Pi_2 \vdash e_2{:}\tau}{\Gamma; \Delta_1 \Cup \Delta_2; \Pi_1 \Cup \Pi_2 \vdash \mathbf{case}\ e\ \mathbf{of}\ \mathbf{inj}_1\ x_1 \mapsto e_2 \mid \mathbf{inj}_2\ x_2 \mapsto e_2 : \tau}$$

$$[\text{T-PACK}]\ \frac{\Gamma; \Delta; \Pi \vdash e : \{\alpha \mapsto \tau\}\tau'}{\Gamma; \Delta; \Pi \vdash \mathbf{pack}\ \alpha{:}\kappa = \tau\ \mathbf{in}\ (e : \tau') : \exists\alpha{:}\kappa.\ \tau'}$$

$$[\text{T-UNPACK}]\ \frac{\Gamma; \Delta_1; \Pi_1 \vdash e_1 : \exists\alpha{:}\kappa.\ \tau \qquad [\Gamma, \alpha{:}\kappa; \Delta_2], x{:}\tau; \Pi_2 \vdash e_2 : \tau'}{\Gamma; \Delta_1 \Cup \Delta_2; \Pi_1 \Cup \Pi_2 \vdash \mathbf{unpack}\ (\alpha, x) = e_1\ \mathbf{in}\ e_2 : \tau'}$$

$$[\text{T-SINK}]\ \Gamma; \cdot; a{\cdot}\rho \vdash \lfloor a \rfloor : \rho \qquad [\text{T-SOURCE}]\ \Gamma; \cdot; a\tilde{\cdot}\rho \vdash \lceil a \rceil : \widetilde{\rho}$$

$$[\text{T-CLOSED}]\ \Gamma; \cdot; a{:}\bot\!\!\!\bot \vdash \lfloor a \rfloor : \bot\!\!\!\bot$$

Figure 3.9: Other expression typing rules for classical F°

$$\S \quad ::= \quad \cdot \;\Big|\; \tilde{\cdot} \;\Big|\; :$$

$$[\text{BP-NIL}]\ [], a\S\rho = [a\S\rho] \qquad [\text{BP-HEAD}]\ \Pi::\overline{\Pi}, a\S\rho = \Pi, a\S\rho::\overline{\Pi}$$

$$[\text{BP-TAIL}]\ \frac{\overline{\Pi}, a\S\rho = \overline{\Pi}'}{\Pi::\overline{\Pi}, a\S\rho = \Pi::\overline{\Pi}'}$$

$$[\text{UC-EMPTY}]\ \cdot \mathbin{\widehat{\cup}} \cdot = \cdot$$

$$[\text{UC-LEFT}]\ \frac{\Pi_1 \mathbin{\widehat{\cup}} \Pi_2 = \Pi \qquad a \notin \mathrm{dom}(\Pi)}{\Pi_1, a\S\rho \mathbin{\widehat{\cup}} \Pi_2 = \Pi, a\S\rho} \qquad [\text{UC-RIGHT}]\ \frac{\Pi_1 \mathbin{\widehat{\cup}} \Pi_2 = \Pi \qquad a \notin \mathrm{dom}(\Pi)}{\Pi_1 \mathbin{\widehat{\cup}} \Pi_2, a\S\rho = \Pi, a\S\rho}$$

$$[\text{UC-SPLIT1}]\ \frac{\Pi_1 \Cup \Pi_2 = \Pi \qquad a \notin \mathrm{dom}(\Pi)}{\Pi_1, a\tilde{\cdot}\rho \mathbin{\widehat{\cup}} \Pi_2, a{\cdot}\rho = \Pi, a{:}\rho} \qquad [\text{UC-SPLIT2}]\ \frac{\Pi_1 \Cup \Pi_2 = \Pi \qquad a \notin \mathrm{dom}(\Pi)}{\Pi_1, a{\cdot}\rho \mathbin{\widehat{\cup}} \Pi_2, a\tilde{\cdot}\rho = \Pi, a{:}\rho}$$

$$[\text{UC*-SINGLE}]\ \widehat{\bigcup}[\Pi] = \Pi \qquad [\text{UC*-SPLIT}]\ \frac{\Pi_1 \mathbin{\widehat{\cup}} \Pi_2 = \Pi \qquad \widehat{\bigcup}\overline{\Pi}_1 = \Pi_1 \qquad \widehat{\bigcup}\overline{\Pi}_2 = \Pi_2}{\widehat{\bigcup}\overline{\Pi}_1, \overline{\Pi}_2 = \Pi}$$

$$[\text{UL-NIL}]\ [] \Cup [] = []$$

$$[\text{UL-LEFT}]\ \frac{\overline{\Pi}_1 \Cup \overline{\Pi}_2 = \overline{\Pi}}{\Pi::\overline{\Pi}_1 \Cup \overline{\Pi}_2 = \Pi::\overline{\Pi}} \qquad [\text{UL-RIGHT}]\ \frac{\overline{\Pi}_1 \Cup \overline{\Pi}_2 = \overline{\Pi}}{\overline{\Pi}_1 \Cup \Pi::\overline{\Pi}_2 = \Pi::\overline{\Pi}}$$

$$[\text{UP-NIL}]\ [] \mathbin{\widehat{\Cup}} [] = [] \qquad [\text{UP-CONS}]\ \frac{\widehat{\bigcup}\overline{\Pi}' = \Pi' \qquad \overline{\Pi}'_1 \Cup \overline{\Pi}'_2 = \overline{\Pi}' \qquad \overline{\Pi}_1 \mathbin{\widehat{\Cup}} \overline{\Pi}_2 = \overline{\Pi}}{\overline{\Pi}'_1, \overline{\Pi}_1 \mathbin{\widehat{\Cup}} \overline{\Pi}'_2, \overline{\Pi}_2 = \Pi'::\overline{\Pi}}$$

Figure 3.10: Auxiliary judgments for classical F° process typing

$$[\text{TP-EXP}]\ \frac{\cdot;\cdot;\Pi;\vdash e : \tau}{[\Pi] \vdash e : \tau} \qquad [\text{TP-NEW}]\ \frac{\overline{\Pi}, a{:}\rho \vdash P : \tau}{\overline{\Pi} \vdash \nu a{:}\rho.\ P : \tau}$$

$$[\text{TP-PARLEFT}]\ \frac{\overline{\Pi}_1 \vdash P_1 : \tau \qquad \overline{\Pi}_2 \vdash P_2 : \bot}{\overline{\Pi}_1 \mathbin{\widehat{\Cup}} \overline{\Pi}_2 \vdash P_1 \mid P_2 : \tau}$$

$$[\text{TP-PARRIGHT}]\ \frac{\overline{\Pi}_1 \vdash P_1 : \bot \qquad \overline{\Pi}_2 \vdash P_2 : \tau}{\overline{\Pi}_1 \mathbin{\widehat{\Cup}} \overline{\Pi}_2 \vdash P_1 \mid P_2 : \tau}$$

Figure 3.11: Process typing rules for classical F°

ing at these leaf rules. A channel context binding may be written $a{\cdot}\rho$, $a\tilde{:}\rho$, or $a{:}\rho$ depending on whether it represents possession of the sink endpoint, the source endpoint, or both endpoints. Only the closed channel $\rfloor a \lfloor$ may be typed with both endpoints, and it must be given the type $\perp$; a binding of $a{\cdot}\rho$ gives $\downarrow a \downarrow$ the type ρ, while $a\tilde{:}\rho$ gives $\uparrow a \uparrow$ the type $\widetilde{\rho}$.

Reasoning about type soundness requires a notion of a well-typed process, written $\overline{\Pi} \vdash P : \tau$, where $\overline{\Pi}$ is a list of channel contexts. Process typing rules are given in Figure 3.11. Rule TP-EXP type checks atomic processes—that is, expressions—with an empty Γ and Δ (as process-level binding involves only channels) and with only a single Π. Rule TP-NEW extends one of the channel contexts in $\overline{\Pi}$ at a ν binder. For parallel composition, recall that every atomic process apart from the original should eventually terminate with some $\rfloor a \lfloor$ of type $\perp$; rules TP-PARLEFT and TP-PARRIGHT thus require that one of their components always have type $\perp$ and therefore give the type of the other.

Figure 3.10 defines the n-ary nondeterministic channel context extension used by TP-NEW as well as the channel context splitting relation $\widehat{\mathbb{U}}$ used by TP-PARLEFT and TP-PARRIGHT. The definition of $\widehat{\mathbb{U}}$ is surprisingly involved, but essentially it entails dividing a list of contexts into two lists of contexts (via $\overline{\mathbb{U}}$), where each context can itself be split (via $\widehat{\cup}$) into additional contexts, with each split of one context into two being accompanied by at most one division of an $a{:}\rho$ binding into an $a{\cdot}\rho$ binding on one side an an $a\tilde{:}\rho$ on the other. The resulting tree structure of context splits ensures that, in any well-typed process of the form $P_1 \mid P_2$, there can be at most one channel for which one endpoint is in P_1 and the other is in P_2. This restriction substantially cuts down the set of well-typed processes and, as will be seen shortly, proves crucial for type soundness.

3.3.1 Standard soundness

Standard soundness is once again defined in terms of preservation (well-typed terms that step always step to well-typed terms) and progress (well-typed non-values can always take a step). In the case of classical F$^\circ$, while preservation is applicable to both expressions and processes, progress can only be a property of well-typed processes, as there are certainly well-typed expressions that require the process evaluation rules in order to take a step. It is worth noting that this standard definition of soundness, by ruling out stuck

terms, guarantees that classical F° programs are free from deadlocks.

Preservation on expressions is a straightforward extension of what was presented in Section 2.3.1:

Lemma 13 (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* $\Gamma_1, \alpha{:}\kappa', \Gamma_2; \Delta; \Pi \vdash e : \tau$ *and* $\Gamma_1 \vdash \tau' : \kappa'$*,*
   then $\Gamma_1, \{\alpha \mapsto \tau'\}\Gamma_2; \{\alpha \mapsto \tau'\}\Delta; \{\alpha \mapsto \tau'\}\Pi \vdash \{\alpha \mapsto \tau'\}e : \{\alpha \mapsto \tau'\}\tau$.

3. *If* $\Gamma_1, x{:}\tau', \Gamma_2; \Delta; \Pi \vdash e : \tau$ *and* $\Gamma_1; \cdot; \cdot \vdash e' : \tau'$*, then* $\Gamma_1, \Gamma_2; \Delta; \Pi \vdash \{x \mapsto e'\}e : \tau$.

4. *If* $\Gamma; \Delta_1, x{:}\tau', \Delta_2; \Pi \vdash e : \tau$ *and* $\Gamma; \Delta'; \Pi' \vdash e' : \tau'$*,*
   then $\Gamma; \Delta_1 \Cup \Delta' \Cup \Delta_2; \Pi \Cup \Pi' \vdash \{x \mapsto e'\}e : \tau$.

Proof. The various cases of substitution are simple extensions of the cases for Lemma 6. The fact that $\widetilde{\rho}$ has kind $\bullet$ whenever ρ does—and that $\widetilde{\rho}$ is not defined otherwise—is crucial for type substitution: if $\alpha^{\perp}$ is accepted as a protocol type, any legal substitution of τ for α must yield a protocol type.

Channel contexts Π are linear and thus are treated analogously to linear contexts Δ, but channels themselves are not variables and are not subject to substitution. □

The obvious extensions to Lemma 7 and Lemma 8 also hold:

Lemma 14. *If* $\Gamma; \Delta; \Pi \vdash v : \tau$ *and* $\Gamma \vdash \tau : \star$*, then* $\Delta = \cdot$ *and* $\Pi = \cdot$.

Proof. Simple induction on the typing derivation. □

Lemma 15 (Permutation)**.** *If* $\Gamma; \Delta; \Pi \vdash e : \tau$*,* Δ' *is a permutation of* Δ*, and* Π' *is a permutation of* Π*, then* $\Gamma; \Delta'; \Pi \vdash e : \tau$.

Proof. Induction on the typing derivation and straightforward reasoning about permutations and the $\Cup$ relation. □

Lemma 16 (Expression preservation). *If* $\Gamma; \Delta; \Pi \vdash e : \tau$ *and* $e \longrightarrow e'$, *then* $\Gamma; \Delta; \Pi \vdash e' : \tau$.

Proof. As Lemma 9; note that E-LAMBDA is the only new evaluation rule that does not operate on the process level, and it clearly preserves typing.

Preservation and progress for processes are more complex—naturally, as most of the evaluation new to classical F° happens at the process level—and must be defined relative to the process equivalence relation $\equiv$. To this end, $\equiv$ must be shown to preserve typing:

Lemma 17 (Preservation under $\equiv$). *If* $\overline{\Pi} \vdash P{:}\tau$ *and* $P \equiv P'$, *then* $\overline{\Pi} \vdash P'{:}\tau$.

Proof. Induction on the process typing derivation; requires associativity of $\hat{\Cup}$. □

Also required is well-typedness of evaluation contexts, written $\Gamma; \Delta; \Pi \vdash E : \left[\tau\right]\tau'$ and defined to be true whenever, for all e, Δ', and Π' such that $\Gamma; \Delta'; \Pi' \vdash e : \tau$, it is the case that $\Gamma; \Delta \Cup \Delta'; \Pi \Cup \Pi' \vdash E\left[e\right] : \tau'$. Reasoning about the communication rules, which change the expression plugged into an evaluation context, requires knowing that the decomposition of a well-typed expression e into $E\left[e'\right]$ always leads to a well-typed E and e' under the appropriate contexts:

Lemma 18 (Well-typed decomposition). *If* $\Gamma; \Delta; \Pi \vdash e : \tau$ *and* $e = E\left[e'\right]$, *then there exist* Δ_1, Δ_2, Π_1, Π_2, *and* τ' *such that* $\Delta = \Delta_1 \Cup \Delta_2$, $\Pi = \Pi_1 \Cup \Pi_2$, $\Gamma; \Delta_1; \Pi_1 \vdash e : \tau'$, *and* $\Gamma; \Delta_2; \Pi_2 \vdash E : \left[\tau'\right]\tau$.

Proof. Induction on the shape of E and reconstructing the typing derivations. □

And, of course, a permutation lemma is required at the process level as well:

Lemma 19 (Process permutation). *If* $\overline{\Pi} \vdash P : \tau$ *and every* Π' *in* $\overline{\Pi}'$ *is a permutation of some unique* Π *in* $\overline{\Pi}$, *then* $\overline{\Pi}' \vdash e : \tau$.

Proof. Induction on the process typing derivation and straightforward reasoning about permutations and the $\hat{\Cup}$ relation. □

(The lemmas so far have been proved—modulo a few basic properties[4], like associativity of $\hat{\mathbb{U}}$—in Coq for Lolliproc, the simply typed version of classical F° (Mazurak and Zdancewic, 2010), and informed the F° definitions of $\equiv$ and $\hat{\mathbb{U}}$.)

Both progress and preservation for classical F° depend on the connection between the restrictions $\hat{\mathbb{U}}$ puts on well-typed processes P and the *communication graph* of P, defined to be the undirected[5] multigraph in which the vertices are the atomic processes (that is, expressions) that make up P and in which an edge exists for each active channel a within the process[6], connecting the expressions containing $\lfloor a \rfloor$ and $\lceil a \rceil$. (No edge exists for $\lfloor a \rfloor$.)

In reasoning about these graphs, it is useful to have a notion of (not necessarily unique) *canonical* processes: a canonical process is of the form $\nu a_1{:}\rho_1. \ldots \nu a_m{:}\rho_m. e_1 \mid (e_2 \mid (\ldots \mid e_n))$ for some $m \geq 0$ and $n \geq 1$. It is easy to see that any process can be made canonical through the process equivalence rules:

Property 20 (Canonicalization)**.** *For any process P, there exists some canonical P' such that $P \equiv P'$.*

Since communication graphs are built out of atomic processes, it is easy to see that this graph structure is invariant under process equivalence—in other words, one can reason about only canonical processes without loss of generality. The remainder of this section will do so liberally.

Property 21 (Graph invariance)**.** *For any processes P and P' where $P \equiv P'$, the communication graph of P' is isomorphic to the communication graph of P.*

Thanks to the exacting requirements of the $\hat{\mathbb{U}}$ relation, there is a correspondence between well-typedness of a process and the acyclicity of its communication graph:

Lemma 22 (Acyclicity and typing)**.** *If $\overline{\Pi} \vdash P : \tau$, then the communication graph of P is acyclic.*

[4] http://www.smbc-comics.com/?id=2204

[5] One might imagine that the directed nature of communication in classical F° would suggest directed graphs, but undirected graphs both entail stronger acyclicity properties and simplify the proof of process preservation.

[6] This definition requires that channel names be globally unique, but note that there is no substitution for channel names or binding of channel names withing expressions to complicate this convention. See also the definition of canonical processes, from which an acceptable renaming can quickly be computed.

Proof. Recall the definition of $\Cup$, which allows only one channel to be split over the two halves of a parallel composition. It is not possible to partition the nodes of a cycle without crossing at least two edges, making it likewise impossible to type check a process with a cyclic communication graph. □

Acyclicity of the communication graph is crucial because deadlocks are essentially cyclic dependencies among concurrent processes. By ruling out cyclic communication entirely, there can be no potentially cyclic dependencies and thus no deadlocks.

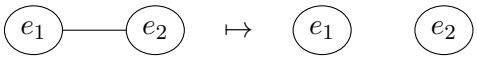
Finally, observe that the acyclicity of the communication graph is preserved under process evaluation:

Lemma 23 (Acyclicity and evaluation)**.** *If* $\overline{\Pi} \vdash P : \tau$ *and there exists* P' *such that* $P \longrightarrow P'$*, then the communication graph of* P' *is also acyclic.*

Proof. With respect to the communication graph of a process, observe that all evaluation steps that respect typing and that alter the graph in any way amount to some combination of the following:

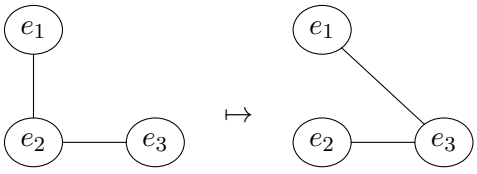
1. the creation of a new vertex and a new edge connecting it to one existing vertex, *e.g.*

2. the deletion of a single edge, *e.g.*



3. the deletion of a single unconnected vertex, *e.g.*

4. and transferring the endpoint of an edge from one vertex to another by sending it across some other edge, *e.g.*



EP-GO entails one instance of (1) along with instances of (4) corresponding to the number of channel endpoints in the argument to $\mathbf{go}^{\rho}$. EP-APPSINK can similarly be seen as a repetition of (4) for every channel endpoint contained in the transferred value, while EP-CLOSE and EP-DONE exactly correspond, respectively, to (2) and (3). All other evaluation rules do not impact the communication graph.

Only (4) could conceivably create a cycle. If a cycle is created, the final step in its creation must be the connection of some atomic processes e_1 and e_2. But this can only be facilitated by some e_3 that is already connected to both e_1 and e_2, in which case a cycle would already exist! Acyclic graphs can thus never become cyclic through these graph operations, and hence a well-typed process—which by Lemma 22 has an acyclic communication graph—will remain acyclic after taking a step. □

Preservation and progress are now within reach—the statements of both lemmas make explicit the idea that process evaluation is always performed modulo the process equivalence relation, though preservation could be stated equally well without $\equiv$.

Lemma 24 (Process preservation)**.** *If* $\overline{\Pi} \vdash P : \tau$ *and there exists some* P' *and* P'' *such that* $P \equiv P'$ *and* $P' \longrightarrow P''$*, then* $\overline{\Pi} \vdash P'' : \tau$*.*

Proof. Observe from Lemma 17 that $\Pi \vdash P' : \tau$ as well, and proceed by induction on this typing derivation and in particular on the expression typing of TP-EXP, the only interesting case. Those subcases that do not follow immediately from Lemma 16 involve or create process structure outside of the current expression; ensuring that they preserve typing means satisfying the requirements of the channel context splitting relation $\widehat{\Cup}$: that at most one $a{:}\rho$ binder be split at each step.

Lemma 23 shows how this is possible: as P' is well-typed, the communication graph of P'' must be acyclic, and from an acyclic communication graph it is always possible to construct a valid $\Cup$ derivation. For EP-TRANSFER, Lemma 18 guarantees that the atomic processes can be successfully decomposed and a value transferred from the sender to the receiver. □

Stating progress requires a precise notion of what it means for a process to be done evaluating. A very simple definition suffices: a process has *finished* if it is equal to $\rfloor a \lfloor$

for some a, or if it contains an atomic process that is a value and that is not $\uparrow a \uparrow$, $\downarrow a \downarrow$, or $\rfloor a \lfloor$—the latter two of which indicate only finished subprocesses, and the former of which should never appear. Once again the canonical forms property of expressions is invoked: all expressions of a given type eventually reduce to that type's introduction form or, for protocol types, to a channel endpoint.

Lemma 25 (Progress)**.** *If $\overline{\Pi} \vdash P : \tau$ where $\tau \neq \bot$, then either P has finished or there exists some P' and P'' such that $P \equiv P'$ and $P' \longrightarrow P''$.*

Proof. Examine each of the atomic processes within P. If, in doing so, an appropriate value or the opportunity to take a step is found, then nothing more is needed; alternatively, an expression e stuck at the elimination of a sink endpoint or a **yield** on a source endpoint may be found. (A lone $\rfloor a \lfloor$ or $\downarrow a \downarrow$ at type $\bot$ may be found in an atomic process, but if the entire process had only these types, that process would have type $\bot$.)

In this stuck case, consider the atomic process e' that contains the other endpoint of the channel in question. If e' itself can take a step, or if e' is ready to communicate with e, then the necessary step has been found. If not e' itself must be stuck at the elimination of a sink endpoint or a **yield** on a source endpoint for some other channel, in which case the search continues.

Because P is well typed, it has an acyclic communication graph, so this search will eventually terminate in the identification of some matching source and sink endpoints that are ready to communicate. To find the exact P' that will step, start at the canonical form of P and repeatedly reassociate atomic processes or push the appropriate channel binding inwards until the process matches the form of one a communication rule. □

Standard soundness follows from progress and preservation:

Theorem 26 (Type soundness)**.** *If $[] \vdash P{:}\tau$, then for any P' and P'' where $P \equiv P'$ and $P' \longrightarrow^* P''$, either and P'' is finished or there exists P''' such that $P'' \longrightarrow P'''$.*

Proof. Induction on $P' \longrightarrow^* P''$ followed by direct application of Lemmas 24 and 25. □

This soundness property guarantees freedom from deadlocks in classical F°. It says nothing, however, about whether an expression will evaluate to a single value or to a

$$\text{[L-SOURCE]}\ \uparrow a \uparrow\ v \xrightarrow[\epsilon]{\epsilon}_j\ v\ (\mathbf{yield}\ \uparrow a \uparrow)$$

$$\text{[L-LAMBDA]}\ \frac{u, z \notin \mathrm{fv}(e)}{\mathbf{yield}\ \lambda^\bullet x{:}\tau \overset{\bullet}{\to} \bot.\ e \xrightarrow[\epsilon]{\epsilon}_j\ \mathbf{let}\ (z,u) = \mathbf{yield}\ (\mathbf{go}^{\tau \overset{\bullet}{\to} \bot}\ \lambda^\bullet x{:}\tau \overset{\bullet}{\to} \bot.\ e)\ \mathbf{in}\ u; z}$$

$$\text{[LP-SINK]}\ \nu a{:}\uparrow\tau\uparrow.\ E\big[\mathbf{yield}\ \downarrow a \downarrow\big] \xrightarrow[\epsilon]{\epsilon}_j\ \nu a{:}\uparrow\tau\uparrow.\ E\big[\mathbf{let}\ (z,u) = \mathbf{yield}\ (\mathbf{go}^{\tau \overset{\bullet}{\to} \bot}\ \downarrow a \downarrow)\ \mathbf{in}\ u; z\big]$$

$$\text{[LP-GO]}\ \frac{a \text{ not free in } E\big[\mathbf{go}^\rho\ v\big]}{E\big[\mathbf{go}^\rho\ v\big] \xrightarrow[\epsilon]{(a:\widetilde{\rho})^j,\ (a:\rho)^{j+1}}_{j+2}\ \nu a{:}\rho.\ (E\big[(a{:}\widetilde{\rho})^j\big]\ \vert\ v\ (a{:}\rho)^{j+1})}$$

$$\text{[LP-TRANSFER]}\ \nu a{:}\tau \overset{\circ}{\to} \rho.\ E_1\big[\mathbf{yield}\ \uparrow a \uparrow\big]\ \vert\ E_2\big[\downarrow a \downarrow\ v\big] \xrightarrow[\epsilon]{\epsilon}_j\ \nu a{:}\rho.\ E_1\big[(v, \uparrow a \uparrow)\big]\ \vert\ E_2\big[\downarrow a \downarrow\big]$$

$$\text{[LP-BRANCH]}\ \nu a{:}\rho_1\ \&\ \rho_2.\ E_1\big[\mathbf{yield}\ \uparrow a \uparrow\big]\ \vert\ E_2\big[\downarrow a \downarrow.i\big] \xrightarrow[\epsilon]{\epsilon}_j\ \nu a{:}\rho_i.\ E_1\big[\mathbf{inj}_i^{\rho_1 \oplus \rho_2}\ \uparrow a \uparrow\big]\ \vert\ E_2\big[\downarrow a \downarrow\big]$$

$$\text{[LP-TTRANSFER]}\ \frac{\nu a{:}\forall\alpha{:}\kappa.\ \rho.\ E_1\big[\mathbf{yield}\ \uparrow a \uparrow\big]\ \vert\ E_2\big[\downarrow a \downarrow\ [\tau]\big] \xrightarrow[\epsilon]{\epsilon}_j}{\nu a{:}\rho.\ E_1\big[\mathbf{pack}\ \alpha{:}\kappa = \tau\ \mathbf{in}\ (\uparrow a \uparrow : \widetilde{\rho})\big]\ \vert\ E_2\big[\downarrow a \downarrow\big]}$$

$$\text{[LP-CLOSE]}\ \nu a{:}\bot.\ E_1\big[\uparrow a \uparrow\big]\ \vert\ E_2\big[\downarrow a \downarrow\big] \xrightarrow[\epsilon]{\epsilon}_j\ E_1\big[()\big]\ \vert\ \nu a{:}\bot.\ E_2\big[\lfloor a \rfloor\big]$$

$$\text{[LP-DONE]}\ P\ \vert\ \nu a{:}\bot.\ \lfloor a \rfloor \xrightarrow[\epsilon]{\epsilon}_j\ P \qquad\qquad \text{[LP-PAR]}\ \frac{P_1 \xrightarrow[D]{C}_k^j\ P_1'}{P_1\ \vert\ P_2 \xrightarrow[D]{C}_k^j\ P_1'\ \vert\ P_2}$$

$$\text{[LP-NEW]}\ \frac{P \xrightarrow[D]{C}_k^j\ P'}{\nu a{:}\tau.\ P \xrightarrow[D]{C}_k^j\ \nu a{:}\tau.\ P'}$$

Figure 3.12: Linearity-aware semantics for classical F°

composition of processes—both are considered acceptable final outcomes, and there is nothing preventing the programmer from, for instance, not matching each call to future with a corresponding call to wait. The next section shows how bringing the annotated semantics into the classical setting addresses this concern by showing that, as long as a program's type is unrestricted, this situation will never occur—evaluation is guaranteed to result in a solitary value.

$$
\begin{array}{rcl}
\mathfrak{T}(\mathbf{go}^{\rho}\ e) & = & \mathfrak{T}(e) \\
\mathfrak{T}(\mathbf{yield}\ e) & = & \mathfrak{T}(e) \\
\mathfrak{T}(\uparrow a \uparrow) & = & \emptyset \\
\mathfrak{T}(\downarrow a \downarrow) & = & \emptyset \\
\mathfrak{T}(\rfloor a \lfloor) & = & \emptyset \\
 & & \\
\mathfrak{T}(P_1 \mid P_2) & = & \mathfrak{T}(P_1) \uplus \mathfrak{T}(P_2) \\
\mathfrak{T}(\nu a{:}\rho.\ P) & = & \mathfrak{T}(P)
\end{array}
$$

Figure 3.13: Tagged contents of new classical F° expressions and processes

3.3.2 Annotated semantics and classical F°

As with intuitionistic F°, one might desire some assurance that linearity delivers what it promises—that is, that linear resources are neither created nor discarded. Luckily, the extension of the annotated semantics from Section 2.3.2 to classical F° is straightforward.

Figure 3.12 gives the annotated semantics for the new constructs in classical F°; because inter-process operations are always concerned with values, and because they reuse the existing machinery of the language, most of these new rules need not be concerned with conscripting or discharging tagged values. The exception is LP-Go: as it creates new channel endpoints, it must conscript them. It would be sound not to conscript sink endpoints here, as they are always immediately passed to the body of the newly created process; conscripting sources is necessary for the proof of Theorem 29, however, so in the interest of consistency LP-Go conscripts both.

The tag contents of the constructs in classical F° (shown in Figure 3.13) are straightforward. It is immediately clear that process equivalence preserves tag contents:

Property 27 (Equivalence of tags)**.** *If $P \equiv P'$, then $\mathfrak{T}(P) = \mathfrak{T}(P')$.*

Recall that a proper expression is one in which unrestricted values do not contain subexpressions tagged with linear types. Extending this to processes, where a proper process contains as atomic processes only proper expressions, it is easy to see that process evaluation also preserves this property:

Lemma 28 (Proper processes)**.** *If P is proper and $P \overset{C}{\underset{D}{\longrightarrow}}{}^{*}{}^{j}_{k}\ P'$, then P' is also proper.*

Proof. Straightforward induction and Lemma 11. The only slightly interesting case is LP-GO, and there is clearly no means by which the tagged channel endpoints it introduces could there be inserted into an unrestricted value. □

Continuing as before, it is not difficult to see that run-time linearity still holds in classical F°:

Theorem 29 (Run time linearity)**.** *If* $\overline{\Pi} \vdash P : \tau$*, P is proper, and* $P \overset{C}{\underset{D}{\longrightarrow}}{}^{*\,j}_{k}\ P'$ *for some* P'*,* C*,* D*,* j*, and* k*, then* $\mathfrak{T}(P)\backslash\star \uplus \{C\}\backslash\star = \mathfrak{T}(P')\backslash\star \uplus \{D\}\backslash\star$*.*

Proof. Straightforward induction as in Theorem 12. □

All this is so straightforward, again, because the tracking of linearity defined by the annotated semantics is essentially unchanged for the new constructs of classical F° and does not interact specially with concurrency. So let us turn to a more interesting question: can an unrestricted computation ever "leak" processes? That is, might an expression of unrestricted type evaluate to a value in parallel with some unfinished processes that will not subsequently evaluate away? Armed with an annotated semantics for classical F°, it can now be proved that the answer is no: every unrestricted expression, no matter what processes are created in it evaluation, will ultimately yield a solitary value.

Theorem 30 (Unrestricted concurrent results)**.** *If* $e \overset{C}{\underset{D}{\longrightarrow}}{}^{*\,j}_{k}\ P$*, P is proper and has finished, and there exists τ such that* $[] \vdash P : \tau$ *and* $\cdot \vdash \tau : \star$*, then either $P = v$ for some v or there exists some P', P'', C', D', and ℓ such that* $P \equiv P'$ *and* $P' \overset{C'}{\underset{D'}{\longrightarrow}}{}^{k}_{\ell}\ P''$*.*

Proof. From the typing rules for processes and our definitions of values and finished processes, it is clear that a finished, well-typed process of type $\tau \neq \bot$—which P is, since it is not the case that $\cdot \vdash \bot : \star$—includes exactly one value v of type τ composed with some number of other processes. If that number is zero, $P = v$, so suppose not.

Since v is proper, it cannot contain any tagged channel endpoints, and, since it is a value of unrestricted type that began as a single expression, it must be that there are no channel endpoints at all within v. Process equivalence is well-behaved for all properties of interest, so, if v contains no channel names, there always exists some P_0 such that $P \equiv v \mid P_0$ and $[] \vdash P_0 : \bot$.

Because P_0 has type $\bot$, it cannot, by definition, be a finished process. P_0 is well-typed, however, so by progress it must be able to take a step. □

3.4 Related work

While intuitionistic linear systems are the most common linear type system in the programming languages literature, linear logic was first presented as a sequent calculus along the lines of similar classical systems. Many have followed that approach, while others have approached classicality (with or without linearity) in the style of natural deduction while differing from F$^\circ$ in other ways. Finally, many typed process calculi exist, though pure process calculi outnumber hybrids like classical F$^\circ$.

3.4.1 Full linear logic

The earliest presentations of linear logic (Girard, 1987; Girard et al., 1989; Abramsky, 1993) are sequent calculi that feature the multiplicative disjunction $\parr$, discussed throughout this chapter, along with the unrestricted modality ! (discussed in Section 2.4), a dual modality ? (sometimes pronounced "why not?" in contrast to "of course!"), and an additive true and false. The $\parr$ connective is in many ways the most intriguing of these; while the other connectives of linear logic—$\otimes$, $\&$, and $\oplus$—now have widely accepted intuitions for programming (Wadler, 2004), $\parr$ eludes both our intuition and our more conventional (according to the programming languages literature) term languages.

Since F$^\circ$ is a classical linear language, it is possible to encode $\parr$ in terms of other connectives; the question remains as to whether any of these encodings will be informative. Starting from the fact that, in a traditional linear sequent calculus, $\sigma_1 \parr \sigma_2 = ((\sigma_1 \parr \sigma_2)^\bot)^\bot$, and recalling that F$^\circ$ mediates between classical and intuitionistic negations, one might define:

$$\rho_1 \parr \rho_2 \triangleq (\widetilde{\rho_1} \otimes \widetilde{\rho_2}) \xrightarrow{\bullet} \bot$$

Or, if $\parr$ should not be restricted to kind $\bullet$:

$$\tau_1 \parr \tau_2 \triangleq ((\tau_1 \xrightarrow{\kappa} \bot) \otimes (\tau_2 \xrightarrow{\kappa} \bot)) \xrightarrow{\bullet} \bot$$

These give a vague intuition that $\parr$ is about joining two sources or process bodies into one but in general do not seem particularly helpful. Another approach is to begin with the definition $\sigma_1 \multimap \sigma_2 \triangleq \sigma_1^{\perp} \parr \sigma_2$. Since $\parr$ is commutative but $\multimap$ (and thus $\overset{\kappa}{\to}$) is not, reversing this definition leads to both $\sigma_1 \parr \sigma_2 \triangleq \sigma_1^{\perp} \multimap \sigma_2$ and $\sigma_1 \parr \sigma_2 \triangleq \sigma_2^{\perp} \multimap \sigma_1$. Being able to go in either of two directions suggest $\&$, leading to two more potential encodings:

$$\begin{aligned} \rho_1 \parr \rho_2 &\triangleq (\widetilde{\rho_1} \overset{\bullet}{\to} \rho_2) \,\&\, (\widetilde{\rho_2} \overset{\bullet}{\to} \rho_1) \\ \tau_1 \parr \tau_2 &\triangleq ((\tau_1 \overset{\circ}{\to} \bot) \overset{\circ}{\to} \tau_2) \,\&\, ((\tau_2 \overset{\circ}{\to} \bot) \overset{\circ}{\to} \tau_1) \end{aligned}$$

These types suggest a stronger intuition for $\parr$: provide a means for dealing with one side of the disjunction and receive the other side. In other words, $\parr$ might be thought of as a suspension of the parallel composition of processes: its elimination requires picking a side for the current process and supplying the other side with a coercion to type $\bot$—appropriately, as all but one atomic process must have type $\bot$. Which encoding—whether one of these two or something else—would be more useful, however, remains to be seen; it is likewise difficult to conceive of how exactly expressions of any of these types could be usefully constructed.

Zeilberger (2006) presents an interesting sequent calculus that, while not actually linear, makes use of the connectives of linear logic for their polarity and gives a term assignment in which eager positive connectives ($\otimes$ and $\oplus$) and lazy negative connectives ($\&$ and $\parr$) coexist harmoniously. The dual calculus (Wadler, 2003, 2012), the symmetric λ-calculus (Filinski, 1989), and the μ-calculus (Curien and Herbelin, 2000) are also tightly tied to sequent calculi while being closer to standard term languages than, *e.g.*, Girard's proof nets. All of these languages define programs as interactions between terms (more familiar expressions that eagerly evaluate to values) and co-terms (variously defined as computations, contexts, or lazy expressions), which still departs rather significantly from what is usually encountered in typed functional programming. The design of F$^\circ$ was highly motivated by a desire to bridge this gap.

3.4.2 Classical natural deduction and control

The connection between term languages for classical logic and control operators has been known for some time (Griffin, 1990; Felleisen and Hieb, 1992; Ong and Stewart, 1997).

The operator **control** from Section 3.1 may be seen as having the type of double-negation elimination ($\forall\alpha.\ ((\alpha \to \bot) \to \bot) \to \alpha$), while the more familiar **callcc** can similarly be given the type of Peirce's Law ($\forall\alpha.\ \forall\beta.\ ((\alpha \to \beta) \to \alpha) \to \alpha$). While these operations cannot be directly imported to a linear setting, they are a significant influence on the design of classical F$^\circ$.

Linear continuations in otherwise unrestricted languages have also been studied, as they indicate certain predictable patterns of control flow (Berdine et al., 2001; Filinski, 1992). Berdine (2004) shows, for example, how such continuations can be used to implement coroutines; classical F$^\circ$ goes further by allowing true concurrent evaluation.

Many (linear or otherwise) natural deduction presentations of classical logics (Parigot, 1992, 1993; Ritter et al., 2000; Bierman, 1999; de Paiva and Ritter, 2006) employ typing judgments with multiple conclusions, *e.g.*:

$$x_1{:}\tau_1, \ldots, x_n{:}\tau_n \vdash e : \tau,\ y_{n+1}{:}\tau_{n+1}, \ldots, y_m{:}\tau_m$$

By duality, such a judgment is logically equivalent to

$$x_1{:}\tau_1, \ldots, x_n{:}\tau_n,\ y_{n+1}{:}\neg\tau_{n+1}, \ldots, y_m{:}\neg\tau_m \vdash e : \tau$$

This approach recovers the usual shape of the typing judgment and so can be reconciled more easily with type systems for functional programming. Moreover, recalling that $\tau \to \bot$ is the type of a continuation accepting τ values, it is possible to read the ys above as binding continuations. Operational semantics in these settings implement commuting conversions, giving rise to some nondeterminism; the correspondence to concurrency is obscured, however, as such semantics rely on the decomposition of a single term (often using evaluation contexts).

The process typing rules of classical F$^\circ$ can be seen as an alternative to this multiple conclusion judgment style. While these systems give all auxiliary conclusions a continuation type $\tau \to \bot$, all but one expression within a classical F$^\circ$ process simply has the type $\perp\!\!\!\perp$. A practical consequence of this design is that, since processes appear only at run time, a type checker for a language based on F$^\circ$ could completely ignore process typing.

3.4.3 Process calculi

Many type systems exist for the π-calculus (Milner et al., 1992), some able to guarantee sophisticated properties—Kobayashi (Kobayashi, 2002) gives a good overview of this area. Many of these type systems use linearity in one form or another (Kobayashi et al., 1999; Beffara, 2006); in fact, session types (Vasconcelos et al., 2006; Caires and Pfenning, 2010; Giunti and Vasconcelos, 2010; Pfenning et al., 2011; Toninho et al., 2013) originated in this setting (Takeuchi et al., 1994; Honda et al., 1998). Sing♯ (Fähndrich et al., 2006) implements many ideas from the process calculus literature; it guarantees, at compile time, that software-isolated processes stay out of each other's way, and further that communication between such processes is not fraught with the possibility of unexpected messages—exactly what classical F$^\circ$ also provides. The complexity of Sing♯, however, presents a high barrier for other languages wishing to adopt its methods.

Programming in a process calculus, however, is also rather different from programming in a traditional functional language, and it is not always clear how to best take ideas from that setting while reusing as much standard machinery as possible. Additionally, type systems for π-calculi are not as tightly coupled to logics as type systems for λ-calculi are, though there has been some work on using π-calculus terms to describe proof reductions (Bellin and Scott, 1994).

The join-calculus (Fournet and Gonthier, 1996; Fournet, 1998) addresses the first of these difficulties with a design built around definitions, defined in parallel, which communicate with each other via defined names. Classical F$^\circ$, by contrast, takes a minimal set of the π-calculus's process constructs and embeds a conventional λ-calculus within them, keeping both the familiarity and logical connections of the latter while acquiring the simple process model of the former.

Chapter 4

Extensions

While the preceding sections have demonstrated the potential of classical F°, it remains a core calculus lacking many features necessary for a full programming language, especially a language with native support for concurrency. This chapter discusses several ways in which F° could be extended in such a direction; none of these extensions appear incompatible with soundness as currently enjoyed by classical F°, implying that programs written with these extensions would still enjoy freedom from deadlocks, resource leaks, and unexpected messages.

4.1 Asynchronous semantics

The communication rules of classical F° are defined synchronously for simplicity's sake, but most languages designed around large numbers of communicating processes (*e.g.*, Erlang (Armstrong et al., 1996) and the join calculus (Fournet and Gonthier, 1996; Fournet, 1998)) do not block on a send until a receiver is ready. Making communication in F° asynchronous[1] does not appear to present any deep theoretical complications.

Languages with and libraries for lightweight processes typically implement a "mailbox" model in which each entity capable of receiving interactions from elsewhere is im-

[1] One might argue that synchronous behavior is what programmers want more often than not, and in a sense this is true; the granularity of the desired synchronization, however, is generally not on the level of suspending execution after a send until the receiver is ready, instead revolving around higher level program logic concerns. At a lower level, the ability to send a message and resume immediately, regardless of the state of the receiver, is highly desirable, much like the ability to return immediately from a write operation regardless of the actual state of any disk or filesystem.

plicitly tied to a queue of messages. Adapting F$^\circ$ to such a model would require keeping track of which operations have already been applied to the sink endpoint of a given channel—the fact that bidirectional communication in F$^\circ$ is encoded in terms of unidirectional communication here serves to simplify what must be added for asynchronicity.

Classical F$^\circ$ currently stores information about a channel at its ν binder, and, thanks to the use of standard elimination forms for operations on sinks, the message queue for a channel can be neatly represented with an evaluation context: the innermost portion of the evaluation context being the head of the queue, the outermost being the tail. Processes in F$^\circ$ extended for asynchronous communication could thus look like

$$P \quad ::= \quad e \;\Big|\; P \parallel P \;\Big|\; \nu E\big[a{:}\rho\big].\, P$$

For a channel bound by $\nu E\big[a{:}\rho\big]$, the type of $\uparrow a \uparrow$ would remain $\widetilde{\rho}$, but the type of $\downarrow a \downarrow$ would now be the type of $E\big[\downarrow a \downarrow\big]$ had $\downarrow a \downarrow$ been of type ρ, thus accounting for the operations that have already been performed on $\downarrow a \downarrow$. For instance, consider a channel a with the protocol type $\rho_1 \mathbin{\&} \rho_2$: if $\downarrow a \downarrow.1$ has been evaluated but $\uparrow a \uparrow$ has not reached its corresponding **yield** statement, the composite process would be of the form

$$\nu\big[a{:}\rho_1 \mathbin{\&} \rho_2\big].1.\, E_1\big[\uparrow a \uparrow\big] \parallel E_2\big[\downarrow a \downarrow\big]$$

Within E_1, $\uparrow a \uparrow$ should still have the type $\uparrow \rho_1 \mathbin{\&} \rho_2 \uparrow$, since this process has not yet reached the next step of the protocol; within E_2, however, $\downarrow a \downarrow$ should have the type ρ_1, reflecting the fact that the selection (now recorded at a's binding site) has already been performed.

Similarly, what was before a single use of EP-TRANSFER would become two evaluation steps on different processes, relying on $\equiv$ to rearrange processes appropriately:

$$\begin{aligned}
\nu E\big[a{:}\tau \overset{\bullet}{\to} \rho\big].\, E'\big[\downarrow a \downarrow v\big] \parallel P \quad &\longrightarrow \quad \nu E\big[a{:}\tau \overset{\bullet}{\to} \rho\big]\; v.\, E'\big[\downarrow a \downarrow\big] \parallel P \\
\nu E\big[(a{:}\tau \overset{\bullet}{\to} \rho)\; v\big].\, E'\big[\textbf{yield}\; \uparrow a \uparrow\big] \parallel P \quad &\longrightarrow \quad \nu E\big[a{:}\rho\big].\, E'\big[(v, \uparrow a \uparrow)\big] \parallel P
\end{aligned}$$

Again, acting on a sink endpoint pushes the action (in this case, the application to v) to the outside of the evaluation context around a's binding site (*i.e.*, the end of the queue), while **yield**ing on a source endpoint uses the innermost portion of that evaluation context (*i.e.*, the head of the queue) in the same manner as in the original synchronous evaluation rules.

Moving to an asynchronous setting would require that TP-NEW invoke expression typing to typecheck its evaluation context, it would add evaluation contexts to the binders

within channel contexts, and it would raise questions for ⓤ—when splitting $E\left[a{:}\rho\right]$ into a source and a sink, should the source simply become $a\tilde{\cdot}\rho$ while the sink becomes $a\cdot\rho'$, where ρ' is the type of $E\left[\lfloor a\rfloor\right]$ had $\lfloor a\rfloor$ been of type ρ? Both of these apparent optimizations eliminate E from Π, so the right course of action likely depends on how the linear resources in E (in particular other channel endpoints) are handled.

On the other hand, perhaps F° should not be extended with primitive asynchronicity at all? It is already possible to simulate asynchronous channels with synchronous one, though not as polymorphically as one might hope: for any given ρ, one can define an asynchronous ago_ρ:

$$\begin{aligned}\mathsf{ago}_\rho &: (\rho \xrightarrow{\bullet} \bot) \xrightarrow{\star} \widetilde{\rho}\\ \mathsf{ago}_\rho &\triangleq \lambda^\star b{:}(\rho \xrightarrow{\bullet} \bot).\ \mathbf{go}^\rho\ \lambda^\bullet snk{:}\rho.\ \mathsf{relay}_\rho\ (\mathbf{go}^\rho\ b)\ snk\end{aligned}$$

Here relay_ρ *src snk* repeatedly **yield**s on *src* and relays the result to *snk* as specified by ρ—for instance, for the protocol type $(\tau \xrightarrow{\bullet} \bot)\ \&\ (\tau \xrightarrow{\bullet} \tau \xrightarrow{\bullet} \bot)$, specifying that the child process will send back either one or two values of type τ, the relay function is

$$\begin{aligned}\mathsf{relay}_{(\tau\xrightarrow{\bullet}\bot)\&(\tau\xrightarrow{\bullet}\tau\xrightarrow{\bullet}\bot)} &: \uparrow\uparrow\tau \otimes \uparrow\mathbf{1}\uparrow\uparrow \oplus \uparrow\tau \otimes \uparrow\tau \otimes \uparrow\mathbf{1}\uparrow\uparrow\uparrow \xrightarrow{\star} (\tau \xrightarrow{\bullet} \bot)\ \&\ (\tau \xrightarrow{\bullet} \tau \xrightarrow{\bullet} \bot) \xrightarrow{\circ} \bot\\ \mathsf{relay}_{(\tau\xrightarrow{\bullet}\bot)\&(\tau\xrightarrow{\bullet}\tau\xrightarrow{\bullet}\bot)} &\triangleq \lambda^\star src{:}\uparrow\uparrow\tau \otimes \uparrow\mathbf{1}\uparrow\uparrow \oplus \uparrow\tau \otimes \uparrow\tau \otimes \uparrow\mathbf{1}\uparrow\uparrow\uparrow.\\ &\qquad \lambda^\circ snk{:}(\tau \xrightarrow{\bullet} \bot)\ \&\ (\tau \xrightarrow{\bullet} \tau \xrightarrow{\bullet} \bot).\\ &\qquad\quad \mathbf{case}\ \mathbf{yield}\ src\ \mathbf{of}\\ &\qquad\qquad \mathbf{inj}_1\ src' \mapsto \mathbf{let}\ snk' = snk.1\ \mathbf{in}\\ &\qquad\qquad\qquad\qquad \mathbf{let}\ (x, u) = \mathbf{yield}\ src'\ \mathbf{in}\ \mathbf{yield}\ u; snk'\ x\\ &\qquad\qquad \mid \mathbf{inj}_2\ src' \mapsto \mathbf{let}\ snk' = snk.2\ \mathbf{in}\\ &\qquad\qquad\qquad\qquad \mathbf{let}\ (y, src'') = \mathbf{yield}\ src'\ \mathbf{in}\\ &\qquad\qquad\qquad\qquad \mathbf{let}\ snk'' = snk'\ y\ \mathbf{in}\\ &\qquad\qquad\qquad\qquad \mathbf{let}\ (z, u) = \mathbf{yield}\ src''\ \mathbf{in}\ \mathbf{yield}\ u; snk''\ z\end{aligned}$$

Ad-hoc polymorphism—*e.g.*, typeclasses as in Haskell (Wadler and Blott, 1989)—could allow relay functions to be defined automatically. A more fundamental issue remains, however, in that encoding asynchronous communication in terms of synchronous communication leads to synchronous and asynchronous channel endpoints with the same type,

potentially leading to confusion when some operations on sink endpoints block and some do not. A mechanism for opaque type aliases (*e.g.*, "newtype") could remedy this, doing so would give to F° channel types distinct from other types, a feature that had so far been avoided.[2] Assuming there is not a compelling use case for synchronous communication, the asynchronous semantics may be preferred despite their additional complexity.

4.2 Recursive types

A natural addition to F°'s existing constructs is *recursive types* of the form $\mu\alpha{:}\kappa.\ \tau$, where any α appearing within τ expands to $\mu\alpha{:}\kappa.\ \tau$. Such types allow for full general recursion, can be used to encode many standard datatypes—*e.g.*, lists. A list can be either linear (with an element type of arbitrary kind) or unrestricted (and able to be duplicated or discarded), but the definition is the same regardless of which kind is chosen:

$$\mathsf{List}_\kappa[\tau] \quad \triangleq \quad \mu\alpha{:}\kappa.\ \mathbf{1} \oplus (\tau \otimes \alpha)$$

Recursive types also enable looping protocols, for which there are many applications—for instance, a session-serving server for sessions of type ρ might have the type $\mu\alpha{:}\bullet.\ \uparrow\rho \otimes \alpha\downarrow$, and a cryptographic random number server with both a synchronous mode (in which the requester must wait until enough entropy is available) and an asynchronous mode (in which a lack of sufficient entropy is announced immediately) could have the type $\mu\alpha{:}\bullet.\ \uparrow\mathsf{Int} \otimes \alpha\downarrow \mathbin{\&} \uparrow\uparrow\mathsf{Int} \otimes \alpha\downarrow \oplus \uparrow\mathbf{1} \otimes \alpha\downarrow\downarrow$

The kinding rule for recursive types acknowledges that the kind annotation on α must be the kind of the entire type, as α expands into that type:

$$[\text{K-REC}]\ \frac{\Gamma, \alpha{:}\kappa \vdash \tau : \kappa}{\Gamma \vdash \mu\alpha{:}\kappa.\ \tau : \kappa}$$

The easiest typing rules are those that support *isorecursive* types, for which the correspondence between α and $\mu\alpha{:}\kappa.\ \tau$ is not a transparent equality but rather is witnessed

[2] Of course, at this point it is entirely unclear whether large scale programming would benefit from unmarked channels; it may be the case that well-designed channel libraries are better off with marked channels. It is perhaps premature, however, to make this decision at the core calculus level.

by $\mathbf{roll}^{\mu\alpha{:}\kappa.\,\tau}$ and **unroll** operations:

$$[\text{T-ROLL}]\ \frac{\Gamma; \Delta; \Pi \vdash e : \{\alpha \mapsto \mu\alpha{:}\kappa.\,\tau\}\tau}{\Gamma; \Delta; \Pi \vdash \mathbf{roll}^{\mu\alpha{:}\kappa.\,\tau}\ e : \mu\alpha{:}\kappa.\,\tau}$$

$$[\text{T-UNROLL}]\ \frac{\Gamma; \Delta; \Pi \vdash e : \mu\alpha{:}\kappa.\,\tau}{\Gamma; \Delta; \Pi \vdash \mathbf{unroll}\ e : \{\alpha \mapsto \mu\alpha{:}\kappa.\,\tau\}\tau}$$

At run time **fold** is simply an eager type constructor eliminated by **unfold**.

It is at present unclear what complications will arise from the interactions between linear and recursive types. The presence of recursive types allows for general recursion, of course—and hence nontermination. Soundness for isorecursive types is usually not difficult to prove, but it remains how such types might interact with the results of Section 3.3.2.

4.3 Nondeterminism

The nondeterminism seen so far in F$^\circ$ has been of an invisible variety—interactions between processes may happen in an undetermined order, but they will all eventually take place—and in fact confluence has been proved for subsets of the F$^\circ$ (Mazurak et al., 2010; Mazurak and Zdancewic, 2010) and assuredly holds for the full language. Real concurrent programming frequently demands more visible nondeterminism, however—one might wish to start several computations in parallel and return whichever finishes first, for example, especially if querying a distributed data structure in which successful searches usually return much more quickly than failed searches.

Such functionality could be obtained with an ambiguous choice operator **amb** of type $\lceil\tau_1\rceil \xrightarrow{\star} \lceil\tau_2\rceil \xrightarrow{\circ} (\tau_1 \xrightarrow{\circ} \lceil\tau_2\rceil \xrightarrow{\circ} \tau') \,\&\, (\tau_2 \xrightarrow{\circ} \lceil\tau_1\rceil \xrightarrow{\circ} \tau') \xrightarrow{\circ} \tau'$, where **amb** $x\ y\ \langle f, g\rangle$ waits on both x and y until one of the two is ready, calling f if x is ready first and g if y is ready first.

Of course, while much useful nondeterminism could be encoded with **amb**, higher level abstractions would be helpful for the case where more than two channel endpoints are in play. For a known, fixed number of channels one might implement a family of

receive functions:

$$\mathsf{receive}_n \quad : \quad \forall\alpha_1{:}\circ.\ \ldots.\ \forall\alpha_n{:}\circ.\ \forall\beta{:}\circ.\ \lceil\alpha_1\rceil \overset{\star}{\to} \ldots \overset{\circ}{\to} \lceil\alpha_n\rceil \overset{\circ}{\to} \\ (\alpha_1 \overset{\circ}{\to} \ldots \overset{\circ}{\to} \lceil\alpha_n\rceil \overset{\circ}{\to} \beta)\ \&\ \ldots\ \&\ (\alpha_n \overset{\circ}{\to} \lceil\alpha_1\rceil \overset{\circ}{\to} \ldots \overset{\circ}{\to} \alpha_{n-1} \overset{\circ}{\to} \beta) \overset{\circ}{\to} \beta$$

Here $\mathsf{receive}_2$ is simply **amb**, while for larger n $\mathsf{receive}_n$ can construct a tree that recaptures "second place" results behind sources as they become available—asynchronicity, be it encoded or via the semantics proposed above, proves helpful here. A call to a receive function thus effectively waits until a **yield** on any of its source arguments can succeed, then selects and applies the appropriate function to handle that result and the other remaining sources. This is reminiscent of the the nondeterministic receive operations found in many concurrent languages—*e.g.*, the join calculus (Fournet and Gonthier, 1996; Fournet, 1998) and Erlang (Armstrong et al., 1996)—but preserves the linearity of channel endpoints.

Of course, these receive functions are not sufficient for every case where nondeterminism is desired—for instance, with recursive types one might have a list containing an unknown number of source endpoints. It is unclear whether **amb** would suffice to encode every nondeterministic operation or if some other primitives may at times be necessary.

4.4 The mix rule

In ordinary, nonlinear logics, $P \wedge Q$ implies $P \vee Q$. In linear logic, $P \,\&\, Q$ implies $P \oplus Q$, but $P \otimes Q$ does not necessarily imply $P \parr Q$. The operation that adds this implication is known in the linear logic literature as the *mix rule*—its addition does not lead to unsoundness, although Girard opposes it due to its effect on the denotational semantics (Girard, 1987).

One can imagine adding an operation **mix** to classical F$^\circ$ using an encoding of $\parr$ like those presented in Section 3.4, *e.g.*:

$$\textbf{mix} \quad : \quad \tau_1 \otimes \tau_2 \overset{\star}{\to} ((\tau_1 \overset{\circ}{\to} \bot) \overset{\circ}{\to} \tau_2)\ \&\ ((\tau_2 \overset{\circ}{\to} \bot) \overset{\circ}{\to} \tau_1)$$

What it would mean to have **mix**? Given values v_1 of type τ_1 and v_2 of types τ_2 along with a sink endpoint s_1 of type $\tau_1 \overset{\bullet}{\to} \bot$, one can now write $(\textbf{mix}\ (v_1, v_2)).1\ \lambda^\circ x{:}\tau_1.\ s_1\ x$, pushing the application of s_1 to v_1 into a separate process and continuing on with v_2. This orphaned child process will send v_1 along s_1 and exit at type $\bot$, but the current process

has meanwhile divested itself of a sink endpoint—and corresponding obligation to return $\bot$ and might well return any arbitrary value into the ether. This corresponds exactly to Girard's observation that the mix rule leads to a separated, non-communicating proof nets.

If such behavior is desired—perhaps in an interactive F$^\circ$ shell or debugger—a simpler alternative to **mix** would be an operation **fork** that directly spawns a new process with no connection to its parent:

$$\textbf{fork} \quad : \quad (\mathbf{1} \overset{\circ}{\to} \tau) \overset{\star}{\to} \mathbf{1}$$

It is not at all clear whether this would ever be desirable outside of a context that provides other means of accessing running processes, however: since these processes can end at arbitrary values, they may leave linear resources (*e.g.*, channel endpoints) suspended in finished processes even when the first process had an unrestricted type, violating the guarantees currently provided by Theorem 30. A safer alternative to **fork**, that accepts only suspended computations of unrestricted type, can already be defined in classical F$^\circ$:

$$\begin{aligned} \mathsf{fork} \quad &: \quad \forall\alpha{:}\star.\, (\mathbf{1} \overset{\circ}{\to} \alpha) \overset{\star}{\to} \mathbf{1} \\ \mathsf{fork} \quad &\triangleq \quad \Lambda\alpha{:}\star.\, \lambda^\star f{:}\mathbf{1} \overset{\circ}{\to} \alpha.\, \mathbf{yield}\ \lambda^\bullet c{:}\bot.\, \mathbf{let}\ \_ = f\ ()\ \mathbf{in}\ c \end{aligned}$$

4.5 Additional kinds

As mentioned in Section 2.4, some languages with substructural types support not just linear types but affine types (which allow discarding but not duplicating) and relevant types (which allow duplicating but not discarding) as well. Combined with unrestricted types, these options suggest a natural four point lattice between $\circ$ and $\star$, with $\bullet$ remaining as it is, since duplicating or discarding a channel endpoint seems incompatible with the sort of correct communication that F$^\circ$ aims to establish. It is unclear whether these additional kinds would be a good use of the complexity budget of a larger language built on F$^\circ$ relative to the inconvenience of using linear types with explicit duplicate or discard operations for those cases where relevant or affine types would be more natural.

Still more can be done by extending F$^\circ$ with higher kinds—particularly the kind constructor $\Rightarrow$ of type-level functions—allowing for quantification over type constructors (which so far has been presented as mere abbreviation). Continuing the reference cell examples

from Section 2.2.2, it becomes possible to 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, resulting in an access capability constructor γ of kind $\star \Rightarrow \circ$:

$$\begin{aligned}\mathsf{SURef}[\tau](\gamma) \triangleq \{\ & \mathsf{set} : \forall\alpha{:}\star.\ \forall\beta{:}\star.\ \beta \xrightarrow{\star} \gamma\ \alpha \xrightarrow{\star} \gamma\ \beta, \\ & \mathsf{get} : \forall\alpha{:}\star.\ \gamma\ \alpha \xrightarrow{\star} \alpha \otimes \gamma\ \alpha, \\ & \mathsf{free} : \forall\alpha{:}\star.\ \gamma\ \alpha \xrightarrow{\star} \mathbf{1}\ \}\end{aligned}$$

$$\mathsf{su\_ref} \ : \ \forall\alpha{:}\star.\ \alpha \xrightarrow{\star} \exists\gamma{:}\star \Rightarrow \circ.\ \gamma\ \alpha \otimes \mathsf{SURef}[\alpha](\gamma)$$

Much as $\mathsf{FH}(\alpha)$ could be coerced to $\mathsf{ROFH}(\alpha)$ in Section 2.2.1, a record of type $\mathsf{SURef}[\tau](\gamma)$ can easily be coerced to type $\mathsf{Ref}[\tau](\gamma\ \tau)$, simply by partial application of member functions. Augmenting SURef along the lines of ShareRef could further allow for shareable reference cells that support strong updates only when they are not shared, a feature found in some capability calculi (Ahmed et al., 2007; Charguéraud and Pottier, 2008).

Section 3.2.2 mentions the difficulty of writing an analog to send and select for sending a type over a channel. The problem, essentially, is that $\forall$ and $\exists$ introduce dependencies at the type level. With $\Rightarrow$, this is no problem at all, though send_type must still be defined at a particular kind κ:

$$\begin{aligned}\mathsf{send\_type}_\kappa \ &: \ \forall\gamma{:}\kappa \Rightarrow \bullet.\ \widetilde{\lceil\exists\alpha{:}\kappa.\ \gamma\ \alpha\rceil} \xrightarrow{\star} \forall\alpha{:}\kappa.\ \widetilde{\gamma\ \alpha} \\ \mathsf{send\_type}_\kappa \ &\triangleq \ \Lambda\gamma{:}\kappa \Rightarrow \bullet.\ \lambda^\star s{:}\widetilde{\lceil\exists\alpha{:}\kappa.\ \gamma\ \alpha\rceil}. \\ &\qquad \mathbf{let}\ (f, u) = \mathbf{yield}\ s\ \mathbf{in} \\ &\qquad\quad u; \Lambda\alpha{:}\kappa.\ \mathbf{go}^{\gamma\ \alpha}\ \lambda^\bullet p{:}\gamma\ \alpha.\ f\ \mathbf{pack}\ \beta{:}\kappa = \alpha\ \mathbf{in}\ (p : \gamma\ \beta)\end{aligned}$$

Higher kinds are a significant addition, however. The metatheory of System F^ω—System F extended with full type-level λs—is quite a bit more complex than that of System F thanks to the introduction of non-trivial equalities between types. Prohibiting type-level λs but allowing constants of higher kind, as is done in ML and Haskell, is one possible compromise, though how much of value this would rule out or enable is unclear.

4.6 Metatheory

This dissertation demonstrates the soundness of F°, both according to the conventional definition (progress and preservation) and with special attention paid to linear resources. Strong normalization—that all expressions evaluate to a value—and confluence—that all evaluations converge—have been proved for subsets of F° (Mazurak and Zdancewic, 2010; Mazurak et al., 2010) and should be true of full classical F°, though the recursive types and nondeterminism proposed above would obviously invalidate these properties.

Parametricity, a property with many uses (Reynolds, 1983; Wadler, 1989), has been proved for the core multiplicative subset of intuitionistic F° (Zhao et al., 2010). It is unclear exactly how parametricity should be formulated for classical F°'s processes, and it would be interesting to see this answered. Parametricity often fails in the presence of side-effects that are invisible at the type level; in this regard, it is uncertain whether process creation and interprocess communication in classical F° would limit the applicability of any parametricity results that can be proved. It might be especially interesting if they do not.

Finally, the connections between F° and other linear logics, especially classical linear logics, remain largely unexplored. While it is conjectured that F° is equivalent in expressivity to many such systems, it would be interesting to see this explored in detail. Doing so would, however, require all the machinery usually associated with translations between sequent calculus and natural deduction formalisms.

Chapter 5

Conclusion

This dissertation has shown that classical F° is sound—that is, that well-typed terms never go wrong—but the novelty of F° is not in its soundness, which, apart from a bit of graph-based reasoning, is relatively standard. Rather, F° demonstrates that linearity can address a wide variety of applications in a way that should appeal to language designers concerned with their complexity budgets: it retains the familiarity of System F and its descendants; it introduces few new constructs, relying primarily on the well-understood connectives of intuitionistic linear logic; and programming in F° retains a straightforward, "follow the type" feel. F° essentially profits from massaging the well-studied underpinnings of linear logic into the familiar forms of typed functional programming

F°'s applicability is demonstrated by its ability to specify protocols at the type level and to do so using standard machinery rather than new constructs specifically added for this purpose. Types in intuitionistic F° can enforce user-specified protocols—*i.e.*, it can encode typestate—through the combination of linearity and polymorphism, while classical F°, by introducing only a handful of source-level constructs, moves, at run time, to a multi-process setting in which communication between processes is simultaneously flexible and type safe—*i.e.*, it can encode session types. There are, of course, protocols that intuitionistic F° cannot capture and aspects of concurrency not addressed by classical F°; it aims not for some notion of feature completeness but to provide significant functionality without onerous overhead.

Soundness for classical F° implies an absence of deadlocks in well-typed classical F°

programs. Starting from a core calculus in which concurrent processes are so well behaved and extending the language to allow new behavior in controlled ways, as described in the preceding chapter, follows a pattern of theoretical exploration that has borne much fruit in the study of function programming. Implementing the features demonstrated by F$^\circ$ in a larger-scale language—be it a language that already exists or a language yet to be created—is no small task, requiring much theoretical work (*e.g.*, verifying the soundness of the various extensions), practical work (*e.g.*, writing and maintaining library code), and reconciliation between theoretical elegance and practical necessity (*e.g.*, any use of channels that cross machine boundaries). Nevertheless, it is my sincere hope that the ideas presented here will serve not just as inspiration for future calculi but as a roadmap to future language implementations that feature substructural types.

Bibliography

Samson Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111:3–57, 1993.

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. ISBN 1-59593-064-7.

Amal Ahmed, Matthew Fluet, and Greg Morrisett. L3: A linear language with locations. *Fundam. Inf.*, 77(4):397–449, 2007. ISSN 0169-2968.

Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang*. Prentice-Hall, 1996.

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

Emmanuel Beffara. A concurrent model for linear logic. *Electronic Notes in Theoretical Computer Science*, 155:147–168, 2006. ISSN 1571-0661.

G. Bellin and P. J. Scott. On the π-calculus and linear logic. *Theoretical Computer Science*, 135(1):11–65, 1994. ISSN 0304-3975.

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

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.

Josh Berdine. *Linear and Affine Typing of Continuation-Passing Style*. PhD thesis, Queen Mary, University of London, 2004.

Josh Berdine, Peter W. O'Hearn, Uday S. Reddy, and Hayo Thielecke. Linearly used continuations. In *Proceedings of the Continuations Workshop*, 2001.

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.

Gavin Bierman. A classical linear lambda calculus. *Theoretical Computer Science*, 227(1–2): 43–78, 1999.

Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. *Information and Computation*, 155(1/2):108–133, November 1999. Special issue of papers from *Theoretical Aspects of Computer Software (TACS 1997)*. An earlier version appeared as an invited lecture in the Third International Workshop on Foundations of Object Oriented Languages (FOOL 3), July 1996.

Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *Proceedings of the 21st International Conference on Concurrency Theory (CONCUR 2010)*, Paris, France, August 2010. Springer LNCS.

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. ISBN 978-1-59593-919-7.

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.

Pierre-Louis Curien and Hugo Herbelin. The duality of computation. In *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming*, September 2000.

Haskell B. Curry, Robert Feys, and William Craig. *Combinatory Logic*, volume 1. North-Holland, Amsterdam, 1958.

Valeria de Paiva and Eike Ritter. A Parigot-style linear λ-calculus for full intuitionistic linear logic. *Theory and Applications of Categories*, 17(3), 2006.

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.

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.

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. ISSN 0163-5980.

M. Felleisen and R. Hieb. A revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992.

Andrzej Filinski. Declarative continuations and categorical duality. Master's thesis, University of Copenhagen, August 1989.

Andrzej Filinski. Linear continuations. In *Proc. 19th ACM Symp. on Principles of Programming Languages (POPL)*, pages 27–38, 1992.

C. Fournet and G. Gonthier. The Reflexive CHAM and the Join-Calculus. In *Proc. ACM Symp. on Principles of Programming Languages (POPL)*, pages 372–385, 1996.

Cédric Fournet. *The Join-Calculus: a Calculus for Distributed Mobile Programming*. PhD thesis, École Polytechnique, nov 1998.

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 *Scandinavian Logic Symposium*, pp. 63–92, North-Holland, 1971.

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

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

Marco Giunti and Vasco T Vasconcelos. A linear account of session types in the pi calculus. In *CONCUR 2010-Concurrency Theory*, pages 432–446. Springer, 2010.

Timothy G. Griffin. A formulae-as-types notion of control. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 47–58. ACM Press, 1990.

Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2):202–220, 2009.

Richard Hickey. Clojure. Available at `http://clojure.org/`, 2007.

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. ISBN 1-58113-945-4.

Graydon Hoare. Rust. Available at `http://www.rust-lang.org/`, 2010.

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.

W. A. Howard. The formulae-as-types notion of contstruction. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Academic Press, 1980.

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. ISBN 978-1-60558-064-7.

Naoki Kobayashi. Type systems for concurrent programs. In *Proceedings of UNU/IIST 10th Anniversary Cooloquium*, March 2002.

Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the Pi-Calculus. *Transactions on Programming Languages and Systems*, 21(5):914–947, 1999.

Yves Lafont. The linear abstract machine. *Theoretical Computer Science*, 59:157–180, 1988. Corrections in vol. 62, pp. 327–328.

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.

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

Karl Mazurak and Steve Zdancewic. Lolliproc: to concurrency from classical linear logic via Curry-Howard and control. In *ICFP '10: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, pages 39–50, New York, NY, USA, 2010. ACM.

Karl Mazurak, Jianzhou Zhao, and Steve Zdancewic. Lightweight linear types in System F°. In *TLDI '10: Proceedings of the 5th ACM SIGPLAN workshop on Types in Language Design and Implementation*, pages 77–88, New York, NY, USA, 2010. ACM.

R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100(1):1–77, 1992.

J. Niehren, J. Schwinghammer, and G. Smolka. A concurrent lambda calculus with futures. *Theor. Comput. Sci.*, 364(3):338–356, 2006. ISSN 0304-3975.

C.-H. L. Ong and C. A. Stewart. A curry-howard foundation for functional computation with control. In *Proc. 24th ACM Symp. on Principles of Programming Languages (POPL)*, pages 215–227, Paris, France, 1997.

Michel Parigot. $\lambda\mu$-calculus: An algorithmic interpretation of classical natural deduction. In *Proceedings of the International Conference on Logic Programming and Automated Reasoning*, volume 624 of *Lecture Notes in Computer Science*, pages 190–201. Springer, 1992.

Michel Parigot. Classical proofs as programs. In *Proceedings of the 3rd Kurt Gödel Colloquium*, volume 713 of *Lecture Notes in Computer Science*, pages 263–276. Springer-Verlag, 1993.

Frank Pfenning, Luis Caires, and Bernardo Toninho. Proof-carrying code in a session-typed process calculus. In *Certified Programs and Proofs*, pages 21–36. Springer, 2011.

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.

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.

Eike Ritter, David J. Pym, and Lincoln A. Wallen. Proof-terms for classical and intuitionistic resolution. *Journal of Logic and Computation*, 10(2):173–207, 2000.

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.

Bernardo Toninho, Luis Caires, and Frank Pfenning. Higher-order processes, functions, and sessions: a monadic integration. In *Programming Languages and Systems*, pages 350–369. Springer, 2013.

Jesse A. Tov and Riccardo Pucella. Practical affine types. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 447–458, New York, NY, USA, 2011a. ACM. ISBN 978-1-4503-0490-0.

Jesse A Tov and Riccardo Pucella. A theory of substructural types and control. In *ACM SIGPLAN Notices*, volume 46, pages 625–642. ACM, 2011b.

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

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.

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. ISBN 978-3-540-85372-5.

Philip Wadler. Theorems for free! In *Proceedings of the 4th International Symposium on Functional Programming and Computer Architecture*, September 1989.

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.

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

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.

Philip Wadler. Call-by-value is dual to call-by-name. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 189–201, New York, NY, USA, 2003. ACM. ISBN 1-58113-756-7.

Philip Wadler. Down with the bureaucracy of syntax! Pattern matching for classical linear logic. unpublished manuscript, 2004.

Philip Wadler. Propositions as sessions. In *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*, pages 273–286. ACM, 2012.

Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76. ACM, 1989.

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.

David Walker. *Advanced Topics in Types and Programming Languages*, chapter 1: Substructural Type Systems. MIT Press, 2005.

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. ISBN 1-58113-095-3.

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.

Noam Zeilberger. On the unity of duality. *Annals of Pure and Applied Logic*, 153(1–3):66–96, 2006.

Jianzhou Zhao, Qi Zhang, and Steve Zdancewic. Relational parametricity for a polymorphic linear lambda calculus. In *Programming Languages and Systems*, pages 344–359. Springer, 2010.

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.