IsoLate: A Type System for Self-recursion

Ravi Chugh

University of Chicago
rchugh@cs.uchicago.edu

Abstract. A fundamental aspect of object-oriented languages is how recursive functions are defined. One semantic approach is to use simple record types and explicit recursion (*i.e.* `fix`) to define mutually recursive units of functionality. Another approach is to use records and recursive types to describe recursion through a "self" parameter. Many systems rely on both semantic approaches as well as combinations of universally quantified types, existentially quantified types, and mixin operators to encode patterns of method reuse, data encapsulation, and "open recursion" through self. These more complex mechanisms are needed to support many important use cases, but they often lack desirable theoretical properties, such as decidability, and can be difficult to implement, because of the equirecursive interpretation that identifies mu-types with their unfoldings. Furthermore, these systems do not apply to languages without explicit recursion (such as JavaScript, Python, and Ruby). In this paper, we present a statically typed calculus of functional objects called IsoLate that can reason about a pattern of mixin composition without relying on an explicit fixpoint operation. To accomplish this, IsoLate extends a standard isorecursive type system with a mechanism for checking the "mutual consistency" of a collection of functions, that is, that all of the assumptions about self are implied by the collection itself. We prove the soundness of IsoLate via a type-preserving translation to a calculus with F-bounded polymorphism. Therefore, IsoLate can be regarded as a stylized subset of the more expressive calculus that admits an interesting class of programs yet is easy to implement. In the future, we plan to investigate how other, more complicated forms of mixin composition (again, without explicit recursion) may be supported by lightweight type systems.

1 Introduction

Researchers have studied numerous foundational models for typed object-oriented programming in order to understand the theoretical and practical aspects of these languages. Many of these models are based on the lambda-calculus extended with combinations of explicit recursion, records, prototype delegation, references, mixins, and traits. Once the dynamic semantics of the language has been set, various type theoretic constructs are then employed in order to admit as many well-behaved programs as possible. These mechanisms include record types and recursive types [7], bounded universal quantification [9], bounded existential quantification [30], F-bounded polymorphism [6,2], and variant parametric types [20,34]. A classic survey by Bruce *et al.* [5] compares many of the core aspects of these systems.

© Springer-Verlag Berlin Heidelberg 2015
J. Vitek (Ed.): ESOP 2015, LNCS 9032, pp. 257–282, 2015.
DOI: 10.1007/978-3-662-46669-8_11

A fundamental aspect of an object calculus is how recursive functions are defined. One option is to include explicit recursion on values (*i.e.* `fix`) as a building block. The evaluation rule

$$\texttt{fix}\,(\lambda f.e) \hookrightarrow e[(\texttt{fix}\,\lambda f.e)/f] \qquad \text{where } e = \lambda x.e'$$

repeatedly substitutes the entire expression as needed, thus realizing the recursion. Explicit recursion is straightforward to reason about: the expression `fix` e has type $S \to T$ as long as e has type $(S \to T) \to (S \to T)$. Similar evaluation and typechecking rules can be defined for recursive non-function values, such as records, using simple syntactic restrictions (see, for example, the notion of *statically constructive* definitions in OCaml [24]). This approach can be used to define objects of (possibly mutually) recursive functions. For example, the following simple object responds to the "message" `f` by multiplying increasingly large integers ad infinitum:

$$\texttt{o1} = \texttt{fix}\,(\lambda\texttt{this}.\{\,\texttt{f} = \lambda\texttt{n}.\texttt{n} * \texttt{this.f(n + 1)}\,\})$$

$$\texttt{o1.f(1)} \hookrightarrow^* \texttt{1} * \texttt{o1.f(2)} \hookrightarrow^* \texttt{1} * \texttt{2} * \texttt{o1.f(3)} \hookrightarrow^* \cdots$$

On the other hand, in a language without explicit recursion on values, recursive computations can be realized by passing explicit "self" (or "this") parameters through function definitions and applications. The following example demonstrates this style:

$$\texttt{o2} = \{\,\texttt{f} = \lambda\texttt{(this,n).n} * \texttt{this.f (this, n + 1)}\,\}$$

$$\texttt{o2.f (o2, 1)} \hookrightarrow^* \texttt{1} * \texttt{o2.f (o2, 2)} \hookrightarrow^* \texttt{1} * \texttt{2} * \texttt{o2.f (o2, 3)} \hookrightarrow^* \cdots$$

Notice that occurrences of `this`, substituted by `o2`, do not require any dedicated evaluation rule to "tie the knot" because the record is explicitly passed as an argument through the recursive calls.

Object encodings using either of the two approaches above — explicit recursion or self-parameter passing — can be used to express many useful programming patterns of "open recursion," including: (1) the ability to define methods independently of their host objects (often referred to as *premethods*) that later get mixed into objects in a flexible way; and (2) the ability to define wrapper functions that interpose on the invocation of methods that have been previously defined.

Compared to explicit recursion, however, the self-parameter-passing approach is significantly harder for a type system to reason about, requiring a combination of equirecursive types, subtyping, and F-bounded polymorphism (where type variable bounds are allowed to be recursive). Such combinations often lack desirable theoretical properties, such as decidability [28,2], and pose implementation challenges due to the equirecursive, or "strong," interpretation of mu-types. Nevertheless, mainstream languages like Java and C# incorporate these features, but this complexity is not suitable for all language designs. For example, the popularity of dynamically typed scripting languages (such as JavaScript, Python, Ruby, and PHP) has sparked a flurry of interest in designing statically typed dialects (such as TypeScript and Hack). It is unlikely that heavyweight mechanisms like F-bounded polymorphism will be easily adopted into such language designs. Instead, it would be useful to have a lightweight type system that could reason about some of the programming patterns enabled by the more complicated systems.

Contributions. The thesis of this paper is that the two patterns of open recursion under consideration do not require the full expressive power of F-bounded polymorphism and equirecursive types. To substantiate our claim, we make two primary contributions:

- We present a statically typed calculus of functional objects called ISOLATE, which is a simple variation and extension of isorecursive types (with only a very limited form of subtyping and bounded quantification). We show that ISOLATE is able to admit an interesting class of recursive programs yet is straightforward to implement. The key feature in ISOLATE is a typing rule that treats records of premethods specially, where all assumptions about the self parameter are checked for mutual consistency.
- To establish soundness of the system, we define a type-preserving translation of well-typed ISOLATE programs into a more expressive calculus with F-bounded polymorphism. As a result, ISOLATE can be regarded as a stylized subset of the traditional, more expressive system.

In languages without `fix` and where the full expressiveness of F-bounded polymorphism is not needed, the approach in ISOLATE provides a potentially useful point in the design space for supporting recursive, object-oriented programming patterns. In future work, we plan to investigate supporting additional forms of mixin composition (beyond what ISOLATE currently supports) and applying these techniques to statically typed dialects of popular scripting languages, which often do not include a fixpoint operator.

Outline. Next, in §2, we provide background on relevant typing mechanisms and identify the kinds of programming patterns we aim to support in ISOLATE. Then, in §3, we provide an overview of how ISOLATE reasons about these patterns in a relatively lightweight way. After defining ISOLATE formally, we describe its metatheory in §4. We conclude in §5 with discussions of related and future work.

2 Background

In this section, we survey how several existing typing mechanisms can be used to define objects of mutually recursive functions, both with and without explicit recursion. We start with a simply-typed lambda-calculus and then extend it with operations for defining records, isorecursive folding and unfolding, and parametric polymorphism. We identify aspects of these systems that motivate our late typing proposal. This section is intended to provide a self-contained exposition of the relevant typing mechanisms; expert readers may wish to skip ahead to §2.3.

Core Language Features and Notation. In Figure 1, we define the expression and type languages for several systems that we will compare in this section. We assume basic familiarity with the dynamic and static semantics for all of these features [29]. In our notation, we define the language $\mathscr{L}$ to be the simply-typed lambda-calculus. We use B to range over some set of base types (`int`, `unit`, `str`, *etc.*) and c to range over constants (`not`, `(*)`, `(++)`, *etc.*).

Base Language $\mathscr{L}$:

$$e ::= c \mid \lambda x.e \mid x \mid e_1\, e_2 \qquad T ::= B \mid T_1 \to T_2$$

Extensions to Expression Language (Denoted by Subscripts):

$$\begin{array}{lrcl}
\mathscr{L}_{\texttt{fix}} & e & ::= & \cdots \mid \texttt{fix}\, e \\
\mathscr{L}_{\{\}} & e & ::= & \cdots \mid \{\, \overline{f} = \overline{e} \,\} \mid e.f \\
\mathscr{L}_{\texttt{iso}} & e & ::= & \cdots \mid \texttt{fold}\, e \mid \texttt{unfold}\, e \\
\mathscr{L}_{\Lambda} & e & ::= & \cdots \mid \Lambda A.e \mid e[T]
\end{array}$$

Extensions to Type Language (Denoted by Superscripts):

$$\begin{array}{lrcl}
\mathscr{L}^{\{\}} & T & ::= & \cdots \mid \{\, \overline{f} : \overline{T} \,\} \\
\mathscr{L}^{\mu} & T & ::= & \cdots \mid A \mid \mu A.T \\
\mathscr{L}^{\forall} & T & ::= & \cdots \mid A \mid \forall A.\, T \\
\mathscr{L}^{\forall A <: T} & T & ::= & \cdots \mid A \mid \forall A <: T.\, T' \mid \texttt{top} \\
\mathscr{L}^{\forall A <: T(A)} & T & ::= & \cdots \mid A \mid \forall A <: T(A).\, T' \mid \texttt{top}
\end{array}$$

Comparison of Selected Language Extensions:

$$\textsc{LangFixSub} \triangleq \mathscr{L}^{\{\},<:,\forall}_{\{\},\texttt{fix},\Lambda} \qquad \textsc{LangMu} \triangleq \mathscr{L}^{\{\},\mu,\forall}_{\{\},\texttt{iso},\Lambda} \qquad \textsc{FSubRec} \triangleq \mathscr{L}^{\{\},\mu=,\forall A <: T(A)}_{\{\},\Lambda}$$

	`fix`	Property A	Property B	Property C	Property D	"Simplicity"
LangFixSub	Y	✓	✓	✓	✓	✓
LangMu	N	✓	✓	×	✓	✓
IsoLate	N	✓	✓	✓	✓⁻	✓
FSubRec	N	✓	✓	✓	✓⁻	×

Fig. 1. Core Languages of Expressions and Types

We define several extensions to the expression and type languages of $\mathscr{L}$. Our notational convention is to use subscripts to denote extensions to the language of expressions and superscripts for extensions to the language of types. In particular, we write $\mathscr{L}_{\texttt{fix}}$, $\mathscr{L}_{\{\}}$, $\mathscr{L}_{\texttt{iso}}$, and $\mathscr{L}_{\Lambda}$ to denote extensions of the base language, $\mathscr{L}$, with the usual notions of fixpoint, records, fold and unfold operators for isorecursive types, and type abstraction and application, respectively. We write $\mathscr{L}^{\{\}}$ to denote the extension of the base type system with record types, $\mathscr{L}^{\mu}$ for isorecursive types, $\mathscr{L}^{\mu=}$ for equirecursive types, $\mathscr{L}^{\forall}$ for (unbounded) universal quantification, $\mathscr{L}^{\forall A <: T}$ for bounded quantification, and $\mathscr{L}^{\forall A <: T(A)}$ for F-bounded quantification (where type bounds can recursively refer to type variables). We attach multiple subscripts or superscripts to denote multiple extensions. For example, $\mathscr{L}^{\{\}}_{\{\},\texttt{fix}}$ denotes the statically typed language with records and a fixpoint operator.

In addition to the syntactic forms defined in Figure 1, we freely use syntactic sugar for common derived forms. For example, we often write $\texttt{let}\, x = e_1\, \texttt{in}\, e_2$ instead of $(\lambda x.e_2)\, e_1$ and we often write $\texttt{let}\, f\, x\, y = e_1\, \texttt{in}\, e_2$ instead of $\texttt{let}\, f = \lambda x.\lambda y.e_1\, \texttt{in}\, e_2$.

We assume the presence of primitive if-expressions, which we write as e_1 `?` e_2 `:` e_3. We also write `val` $x :: T$ as a way to ascribe an expected type to a let-bound expression.

Comparison of Systems. We define aliases in Figure 1 for three systems to which we will pay particular attention. We write LangFixSub to refer to the language of records, explicit recursion, and subtyping; LangMu to refer to the language of records and isorecursive mu-types; and FSubRec to refer to the language of records, equirecursive mu-types, and F-bounded polymorphism. In addition, each of these systems has universal quantification, which is unbounded in LangFixSub and LangMu and F-bounded in FSubRec. Notice that LangMu and FSubRec do not include explicit recursion, indicated by the absence of `fix` in the subscripts. The name FSubRec is a mnemonic to describe the often-called System Fsub extended with recursive bounds and recursive types.

Next, we will compare these three languages. The table at the bottom of Figure 1 summarizes their differences along a number of dimensions that we will discuss. Properties A through D are four programming patterns that are of interest for this paper, and "Simplicity" informally refers to implementation and metatheoretic challenges that the type system presents. Then, in § 3, we will explain how our IsoLate calculus identifies a new point in the design space that fits in between LangMu and FSubRec.

2.1 LangFix and LangFixSub: Recursion with `fix`

We will start with the language LangFix $\triangleq \mathscr{L}^{\{\}}_{\{\},\texttt{fix}}$ of records and explicit recursion (without subtyping), in which mutually recursive functions can be defined as follows:

$$\texttt{Ticker} \triangleq \texttt{\{ tick:int} \to \texttt{str; tock:int} \to \texttt{str \}}$$

```
val ticker0 :: Ticker
let ticker0 = fix \this.
  let f n = n > 0 ? "tick " ++ this.tock (n)   : "" in
  let g n = n > 0 ? "tock " ++ this.tick (n-1) : "" in
  { tick = f; tock = g }

ticker0.tick 2  -- "tick tock tick tock "
```

As mentioned in § 1, typechecking expressions of the form `fix` e is simple. We will build on this example to demonstrate several programming patterns of interest.

[Property A] Defining Premethods Separately. In the program above, all components of the mutually recursive definition appear together (*i.e.* syntactically adjacent) inside the fixpoint expression. For reasons of organization, the programmer may want to instead structure the component definitions separately and then combine them later:

```
val tick, tock :: Ticker -> int -> str

let tick this n = n > 0 ? "tick " ++ this.tock (n)   : ""
let tock this n = n > 0 ? "tock " ++ this.tick (n-1) : ""
```

```
let ticker1 = fix \this.
  let f n = tick this n in
  let g n = tock this n in
  { tick = f; tock = g }

ticker1.tick 2  -- "tick tock tick tock "
```

Furthermore, if the programmer wants to define a second quieter ticker that does not emit the string "`tock`", defining the component functions separately avoids the need to duplicate the implementation of `tick`:

```
val tock' :: Ticker -> int -> str
let tock' this n = this.tick (n-1)

let ticker2 = fix \this.
  let f n = tick this n in
  let g n = tock' this n in
  { tick = f; tock = g }

ticker2.tick 2  -- "tick tick "
```

Notice that the implementations of `tick`, `tock`, and `tock'` lay outside of the recursive definitions `ticker1` and `ticker2` and are each parameterized by a `this` argument. Because these three functions are not inherently tied to any record, we refer to them as *premethods*. In contrast, we say that the functions `f` and `g` in `ticker1` (respectively, `ticker2`) are *methods* of `ticker1` (respectively, `ticker2`) because they are tied to that particular object.[1] The method definitions are eta-expanded to ensure that the recursive definitions are syntactically well-founded (*e.g.* [24]).

[Property B] Intercepting Recursive Calls. Another benefit of defining premethods separately from their eventual host objects is that it facilitates "intercepting" recursive calls in order to customize behavior. For example, say the programmer wants to define a louder version of the ticker that emits exclamation points in between each "`tick`" and "`tock`". Notice that the following reuses the premethods `tick` and `tock` from before:

```
let ticker3 = fix \this.
  let f n = "! " ++ tick this n in
  let g n = "! " ++ tock this n in
  { tick = f; tock = g }

ticker3.tick 2  -- "! tick ! tock ! tick ! tock ! "
```

[1] The term premethod is sometimes used with a slightly different meaning, namely, for a lambda that closes over an implicit receiver variable rather than than explicitly taking one as a parameter. We use the term premethod to emphasize that the first (explicit) function argument is used to realize recursive binding.

[Property C] Mixing Premethods into Different Types. Having kept premethod definitions separate, the programmer may want to include them into objects with various types, for example, an extended ticker type that contains an additional boolean field to describe whether or not its volume is loud:

$$\texttt{TickerVol} \triangleq \{\ \texttt{tick:int} \to \texttt{str};\ \texttt{tock:int} \to \texttt{str};\ \texttt{loud:bool}\ \}$$

Intuitively, the mutual requirements between the `tick` and `tock'` premethods are independent of the presence of the `loud` field, so we would like the type system to accept the following program:

```
let quietTicker = fix \this.
  let f n = tick this n in
  let g n = tock' this n in
  { tick = f; tock = g; loud = false }
```

This program is not type-correct in LANGFIX, because the types derived for `tick` and `tock'` pertain to `Ticker` rather than `TickerVol`. Extending LANGFIX with the usual notion of record subtyping, resulting in a system called LANGFIXSUB, addresses the problem, however.

In addition, LANGFIXSUB can assign the following less restrictive types to the same premethods from before: $\texttt{tick} :: \texttt{Tock} \to \texttt{int} \to \texttt{str}$, $\texttt{tock} :: \texttt{Tick} \to \texttt{int} \to \texttt{str}$, and $\texttt{tock'} :: \texttt{Tick} \to \texttt{int} \to \texttt{str}$. Notice that the types of the `this` arguments are described by the following type abbreviations, rather than `Ticker`, to require only those fields used by the definitions:

$$\texttt{Tick} \triangleq \{\ \texttt{tick:int} \to \texttt{str}\ \} \qquad \texttt{Tock} \triangleq \{\ \texttt{tock:int} \to \texttt{str}\ \}$$

[Property D] Abstracting Over Premethods. The last scenario that we will consider using our running example is abstracting over premethods. For example, the following wrapper functions avoid duplicating the code to insert exclamation points in the definition of `ticker3` from before:

```
val wrap :: all A,B,C,C'. (C->C') -> (A->B->C) -> (A->B->C')
let wrap g f x y = g (f x y)

val exclaim :: all A,B. (A -> B -> str) -> (A -> B -> str)
let exclaim = wrap _ _ _ _ (\s. "! " ++ s)

let ticker3' = fix \this.
  let f n = (exclaim _ _ tick) this n in
  let g n = (exclaim _ _ tock) this n in
  { tick = f; tock = g }

ticker3'.tick 2  -- "! tick ! tock ! tick ! tock ! "
```

The two calls to `exclaim` (and, hence, `wrap`) are made with two different premethods as arguments. Because these premethods have different types in LANGFIXSUB, (unbounded) parametric polymorphism is required for typechecking. Note that we write underscores where type instantiations can be easily inferred.

"Open" vs. "Closed" Objects. As we have seen throughout the previous examples, the type of a record-of-premethods differs from that of a record-of-methods. We refer to the former kind of records as *open objects* and the latter as *closed objects*. Once closed, there is no way to extract a method to be included into a closed object of a different type. The following example highlights this distinction, where the `makeLouderTicker` function takes a record of two premethods and wraps them before creating a closed `Ticker` object:

$$\texttt{PreTicker} \triangleq \{\, \texttt{tick:Tock} \to \texttt{int} \to \texttt{str};\ \texttt{tock:Tick} \to \texttt{int} \to \texttt{str} \,\}$$

```
val makeLouderTicker :: PreTicker -> Ticker
let makeLouderTicker openObj = fix \closedObj.
  let f n = (exclaim _ _ openObj.tick) closedObj n in
  let g n = (exclaim _ _ openObj.tock) closedObj n in
  { tick = f; tock = g }

let (ticker4, ticker5) =
  ( makeLouderTicker { tick =              tick; tock = tock }
  , makeLouderTicker { tick = exclaim _ _ tick; tock = tock } )

ticker4.tick 2  -- "! tick ! tock ! tick ! tock ! "
ticker5.tick 2  -- "! ! tick ! tock ! ! tick ! tock ! ! "
```

The first row of the table in Figure 1 summarizes that LANGFIXSUB supports the four programming scenarios outlined in the previous section. Next, we will consider how the same scenarios manifest themselves in languages *without* an explicit `fix`. Such encodings may be of theoretical interest as well as practical interest for object-oriented languages such as JavaScript, in which the semantics does not include `fix`.

2.2 LANGMU: Recursion with Mu-Types

We will consider the language LANGMU of records, isorecursive mu-types, and unbounded universal quantification. The standard rules for isorecursive types are:

$$\frac{T = \mu A.S \qquad \Gamma \vdash e : S[T/A]}{\Gamma \vdash \texttt{fold}\, T\, e : T} \qquad \frac{T = \mu A.S \qquad \Gamma \vdash e : T}{\Gamma \vdash \texttt{unfold}\, e : S[T/A]}$$

We define the syntactic sugar $e\,\texttt{\$}\,f\,(e') \triangleq (\texttt{unfold}\, e).f\,(e)\,(e')$ to abbreviate the common pattern of unfolding a recursive record, reading a method stored in one of its fields, and then calling the method with the (folded) record as the receiver (first argument) to the method.

[Properties A and B]. Defining premethods separately and interposing on recursive calls are much the same in LANGMU as they are in LANGFIXSUB. Using the typechecking rules for isorecursive types above, together with the usual rule for function application, we can write the following in LANGMU:

$$\texttt{Ticker} \triangleq \mu A.\{\,\texttt{tick:}A \to \texttt{int} \to \texttt{str};\ \texttt{tock:}A \to \texttt{int} \to \texttt{str}\,\}$$

```
val tick, tock, tock' :: Ticker -> int -> str
let tick this n  = n > 0 ? "tick " ++ this$tock(n)   : ""
let tock this n  = n > 0 ? "tock " ++ this$tick(n-1) : ""
let tock' this n = this$tick(n-1)

let wrap g f x y = g (f x y)
let exclaim = wrap (\s. "! " ++ s)

let (ticker1, ticker2, ticker3) =
  ( fold Ticker { tick = tick         ; tock = tock         }
  , fold Ticker { tick = tick         ; tock = tock'        }
  , fold Ticker { tick = exclaim tick ; tock = exclaim tock })

ticker1$tick(2)  -- "tick tock tick tock "
ticker2$tick(2)  -- "tick tick "
ticker3$tick(2)  -- "! tick ! tock ! tick ! tock ! "
```

[Property D]. As in LANGFIXSUB, unbounded universal quantification in LANGMU can be used to give general types to functions, such as `wrap` and `exclaim` above, that abstract over premethods.

[Property C]. Premethods in LANGMU cannot be folded into *different* mu-types than the ones specified by the annotations for their receiver arguments. Consider the following example that attempts to, as before, define an extended ticker type that contains an additional boolean field:

$$\texttt{TickerVol} \triangleq \mu A.\{\,\texttt{tick:}A \to \texttt{int} \to \texttt{str};\ \texttt{tock:}A \to \texttt{int} \to \texttt{str};\ \texttt{loud:bool}\,\}$$

```
let quietTicker =
  fold TickerVol { tick = tick ; tock = tock' ; loud = false }
```

The problem is that the record type

$$\{\,\texttt{tick,tock:Ticker} \to \texttt{int} \to \texttt{str};\ \texttt{loud:bool}\,\}$$

does not equal the unfolding of `TickerVol`

$$\{\,\texttt{tick,tock:TickerVol} \to \texttt{int} \to \texttt{str};\ \texttt{loud:bool}\,\}$$

as required by the typing rule for `fold`. In particular, $\texttt{Ticker} \neq \texttt{TickerVol}$. Unlike for LANGFIX, simply adding subtyping to the system does not address this difficulty. In the above record type comparison, the contravariant occurrence of the mu-type would require that `TickerVol` be a subtype of `Ticker`, which seems plausible by record width subtyping. However, the standard "Amber rule"

$$\frac{\Gamma, A_1 <: A_2 \vdash T_1 <: T_2}{\Gamma \vdash \mu A_1.T_1 <: \mu A_2.T_2}$$

for subtyping on mu-types requires that the type variable vary covariantly [8,29]. As a result, $\texttt{TickerVol} \not<: \texttt{Ticker}$ which means that the code snippet fails to typecheck even in LANGMU extended with subtyping.

The only recourse in LANGMU is to duplicate premethods multiple times, one for each target mu-type. The second row of the table in Figure 1 summarizes the four programming scenarios in the context of LANGMU, using an $\times$ to mark that Property C does not hold.

2.3 FSUBREC: Recursion with F-bounded Polymorphism

Adding subtyping to LANGMU is not enough, on its own, to alleviate the previous limitation, but it is when combined with a more powerful form of universal quantification. In particular, the system we need is FSUBREC, which contains (1) F-bounded polymorphism, where a bounded universally quantified type $\forall A <: S.\, T$ allows its type bound S to refer to the type variable A being constrained; and (2) equirecursive, or "strong" recursive, types where a mu-type $\mu A.T$ is considered definitionally equal to its unfolding $T[(\mu A.T)/A]$ in all contexts. That means that there are no explicit `fold` and `unfold` operations in FSUBREC as there are in languages with isorecursive, or "weak" recursive, types like LANGMU.

To see why "weak recursion is not a good match for F-bounded quantification," as described by Baldan *et al.* [2], consider the type instantiation rule

$$\frac{\Gamma \vdash e : \forall A <: S.\, T \qquad \Gamma \vdash S' <: S[S'/A]}{\Gamma \vdash e\,[S'] \,:\, T[S'/A]}$$

which governs how bounded universals can be instantiated. Notice that the particular type parameter S' must be a subtype of the bound S where all (recursive) occurrences of A are replaced with S' itself. A recursive type can satisfy an equation like this only when it is considered definitionally equal to its unfolding, because the structure of the types S and S' simply do not match (in all of our examples, S' is a mu-type but S is a record type).

[Properties A and C]. Having explained the motivation for including equirecursive types in FSUBREC, we return to our example starting with premethod definitions:

$$\begin{aligned}
\texttt{Tick}(A) &\triangleq \{\ \texttt{tick}{:}A \to \texttt{int} \to \texttt{str}\ \} \\
\texttt{Tock}(A) &\triangleq \{\ \texttt{tock}{:}A \to \texttt{int} \to \texttt{str}\ \} \\
\texttt{TickPre}(B,C) &\triangleq (\forall A <: \texttt{Tick}(A).\, A \to B \to C) \\
\texttt{TockPre}(B,C) &\triangleq (\forall A <: \texttt{Tock}(A).\, A \to B \to C)
\end{aligned}$$

```
val tick        :: TockPre (int, str)
val tock, tock' :: TickPre (int, str)

let tick this n  = n > 0 ? "tick " ++ this.tock(this)(n)   : ""
let tock this n  = n > 0 ? "tock " ++ this.tick(this)(n-1) : ""
let tock' this n = this.tick(this)(n-1)
```

There are two aspects to observe. First, the premethod types, which use F-bounded universals, require that the `this` parameters have only the fields used by the definitions (like in LANGFIXSUB). Second, the implementations of `tick`, `tock`, and `tock'` do not include `unfold` expressions (unlike in LANGMU), because the type system implicitly folds and unfolds equirecursive types as needed.

We can now mix the premethods into various target mu-types, such as `Ticker` and `TickerVol` as defined in LANGMU, by instantiating the F-bounded universals appropriately. In the following, we use square brackets to denote the application, or instantiation, of an expression to a particular type.

```
let (normalTicker, quietTicker) =
 ( { tick=tick[TickerVol]; tock=tock [TickerVol]; loud=false }
 , { tick=tick[TickerVol]; tock=tock'[TickerVol]; loud=false } )

normalTicker.tick(normalTicker)(2)  -- "tick tock tick tock "
quietTicker.tick(quietTicker)(2)    -- "tick tick "
```

[Property B]. Interposing on recursive calls in FSUBREC is much the same as before:

```
val ticker3 :: Ticker
let ticker3 =
  let f this n = "! " ++ tick [Ticker] this n in
  let g this n = "! " ++ tock [Ticker] this n in
  { tick = f; tock = g }

ticker3.tick 2  -- "! tick ! tock ! tick ! tock ! "
```

[Property D]. Abstracting over premethods in FSUBREC, however, comes with a caveat. Symptoms of the issue appear in the definitions of `f` and `g` in `ticker3` above: notice that the `tick` and `tock` premethods are instantiated to a particular type and then wrapped. As a result, `f` and `g` are *methods* tied to the `Ticker` type rather than *premethods* that can work with various host object types. We can abstract the wrapper code in `ticker3` as in LANGFIXSUB and LANGMU, but the fact remains that `wrap` and `exclaim` below operate on methods rather than premethods:

```
val wrap :: all A,B,C,C'. (C->C') -> (A->B->C) -> (A->B->C')
let wrap g f x y = g (f x y)
let exclaim = wrap _ _ (\s. "! " ++ s)

let loudTicker =
  { tick = exclaim _ _ (tick [TickerVol])
  ; tock = exclaim _ _ (tock [TickerVol])
  ; loud = true }

loudTicker.tick(loudTicker)(2)
  -- "! tick ! tock ! tick ! tock ! "
```

If we wanted to define a function `exclaim'` that truly abstracts over premethods (that is, which could be called with the uninstantiated `tick` and `tock` values), the type of `exclaim'` must have the form

$$\forall R.\ (\forall A <: R.\ A \to \texttt{int} \to \texttt{str}) \to (\forall A <: R.\ A \to \texttt{int} \to \texttt{str})$$

so that the type variable R could be instantiated with $\texttt{Tock}(A)$ or $\texttt{Tick}(A)$ as needed at each call-site. This kind of type cannot be expressed in FSUBREC, however, because these type instantiations need to refer to the type variable A which is not in scope.

As a partial workaround, the best one can do in FSUBREC is to define wrapper functions that work only for particular premethod types and then duplicate definitions for different types. In particular, we can specify two versions of the type signatures

$$\begin{aligned}
\texttt{wrapTick} &:: \forall B, C, C'.\ \texttt{TickPre}(B, C) \to \texttt{TickPre}(B, C') \\
\texttt{wrapTock} &:: \forall B, C, C'.\ \texttt{TockPre}(B, C) \to \texttt{TockPre}(B, C') \\
\texttt{exclaimTick} &:: \texttt{TickPre}(\texttt{int}, \texttt{str}) \to \texttt{TickPre}(\texttt{int}, \texttt{str}) \\
\texttt{exclaimTock} &:: \texttt{TockPre}(\texttt{int}, \texttt{str}) \to \texttt{TockPre}(\texttt{int}, \texttt{str})
\end{aligned}$$

and then define two versions of the wrappers as follows:

```
let wrapTick B C C' g f x y = \A. g (f[A] x y)
let wrapTock B C C' g f x y = \A. g (f[A] x y)

let exclaimTick = wrapTick _ _ _ (\s. "! " ++ s)
let exclaimTock = wrapTock _ _ _ (\s. "! " ++ s)

let loudTicker' =
  { tick = (exclaimTock tick) [TickerVol]
  ; tock = (exclaimTick tock) [TickerVol]
  ; loud = true }

loudTicker'.tick(loudTicker')(2)
  -- "! tick ! tock ! tick ! tock ! "
```

With this approach, the wrapper functions take premethods as inputs and return premethods as outputs. This code duplication is undesirable, of course, so in the FSUBREC row of the table in Figure 1 we qualify the check mark for Property D with a minus sign.

***Undecidability of* FSUBREC.** Equirecursive types and F-bounded polymorphism are powerful, indeed, which is why they are often used as the foundation for object calculi, sometimes with additional constructs like type operators and bounded existential types [5]. This power comes at a cost, however, both in theory and in practice. Subtyping for System Fsub (*i.e.* bounded quantification) is undecidable, even when type bounds are not allowed to be recursive [28], and the addition of equirecursive types poses challenges for the completeness of the system [16,2]. There exist decidable fragments of System Fsub that avoid the theoretically problematic cases without affecting

many practical programming patterns. However, mainstream languages (*e.g.* Java and C#) include features beyond generics and subtyping such as variance, and the subtle interaction between these features is an active subject of research (*e.g.* [23,17]). Furthermore, an equirecursive treatment of mu-types demands more work by the type checker — treating recursive types as graphs and identifying equivalent unfoldings — than does isorecursive types, where the recursive type operator is "rigid" and, thus, easy to support [29]. As a result of these complications, we mark the "Simplicity" column for FSUBREC in Figure 1 with an $\times$. In settings where the full expressive power of these features is needed, then the theoretical and practical hurdles that accompany them are unavoidable. But for other settings, ideally we would have a more expressive system than LANGMU that is much simpler than FSUBREC.

3 The ISOLATE Calculus

We now present our calculus, ISOLATE, that aims to address this goal. Our design is based on two key observations about the FSUBREC encodings from the previous section: first, that the record types used to describe self parameters mention only those fields actually used by the function definitions; and second, that when creating an object out of a record of premethods, each premethod is instantiated with the mu-type that describes the resulting object.

The ISOLATE type system includes special support to handle this common programming pattern without providing the full expressive power, and associated difficulties, of FSUBREC. Therefore, as the third row of the table in Figure 1 outlines, ISOLATE satisfies the same Properties A through D as FSUBREC but fares better with respect to "Simplicity," in particular, because it is essentially as easy as LANGMU to implement.

3.1 Overview

Before presenting formal definitions, we will first work through an ISOLATE example split across Figure 2, Figure 3, and Figure 4.

Premethods. Let us first consider the premethod definitions of `tick`, `tock`, and `tock'` on lines 1 through 8, which bear many resemblances to the versions in FSUBREC. The special *pre-type* $(A:S) \Rightarrow T$ in ISOLATE is interpreted like the type $\forall A<:S.\, A \to T$ in FSUBREC. Values that are assigned pre-types are special functions called *premethods* $\varsigma x:A<:S.e$, which are treated like polymorphic function values $\Lambda A<:S.\lambda x:A.e$ in FSUBREC. A notational convention that we use in our examples is that the identifier `this` signifies that the enclosing function desugars to a premethod rather than an ordinary lambda. Notice that the types `Tick(A)` and `Tock(A)` describe only those fields that are referred to explicitly in the definitions. A simple rule for unfolding self parameters allows the definitions of `tick`, `tock`, and `tock'` to typecheck.

Closing Open Records. ISOLATE provides special support for sending messages to records of premethods. To keep subsequent definitions more streamlined, in ISOLATE we require that all premethods take two arguments (in curried style). Therefore, instead

```
1  type Tick(A)          =  { tick: A -> int -> str }
2  type Tock(A)          =  { tock: A -> int -> str }
3
4  val tick              :: (A:Tock(A)) => int -> str
5  val tock, tock'       :: (A:Tick(A)) => int -> str
6  let tick this n       =  n > 0 ? "tick " ++ this$tock(n)   : ""
7  let tock this n       =  n > 0 ? "tock " ++ this$tick(n-1) : ""
8  let tock' this n      =  this$tick(n-1)
9
10 val const             :: bool -> (A:{}) => unit -> bool
11 let const b this ()   =  b
12 let (true_, false_)   =  (const true, const false)
13
14 let normalTicker      =  { tick = tick ; tock = tock  ; loud = false_ }
15 let quietTicker       =  { tick = tick ; tock = tock' ; loud = false_ }
16
17 normalTicker # tick(2)    -- "tick tock tick tock "
18 quietTicker  # tick(2)    -- "tick tick "
```

Fig. 2. ISOLATE Example (Part 1)

of using boolean values `true` and `false` to populate a `loud` field of type `bool`, on line 12 we define premethods `true_` and `false_` of the following type, where the type $Bool \triangleq (A{:}\{\,\}) \Rightarrow \texttt{unit} \to \texttt{bool}$ imposes no constraints on its receivers.

Having defined the required premethods, the expressions on lines 14 through 18 show how to build and use records of premethods. The definitions of `normalTicker` and `quietTicker` create ordinary records described by the record type

$$R_0 \triangleq \texttt{OpenTickerVol} \triangleq \{\,\texttt{tick:}\mathit{PreTick}\texttt{; tock:}\mathit{PreTock}\texttt{; loud:}\mathit{Bool}\,\}$$

where we make use of abbreviations $\mathit{PreTick} \triangleq (A{:}\texttt{Tock}(A)) \Rightarrow \texttt{int} \to \texttt{str}$ and $\mathit{PreTock} \triangleq (A{:}\texttt{Tick}(A)) \Rightarrow \texttt{int} \to \texttt{str}$. We refer to these two records in ISOLATE as "open" because they do not yet form a coherent "closed" object that can be used to invoke methods. The standard typing rule for `fold` e expressions does not apply to these open records, because the types of their receivers do not match (as was the difficulty in our LANGMU example). To use open objects, ISOLATE provides an additional expression form `close` e and the following typing rule:

$$\frac{\Gamma \vdash e : R \qquad \mathsf{Guar}_A(R) \supseteq \mathsf{Rely}_A(R) \qquad T = \mu A.\mathsf{Coerce}(\mathsf{Guar}_A(R))}{\Gamma \vdash \texttt{close}\, e : T}$$

The rule checks that an open record type R of premethods is mutually consistent (the second premise) and then freezes the type of the resulting record to be exactly what is guaranteed (the third premise). To define mutual consistency of a record R, we introduce the notions of *rely-set* and *guarantee-set* for each pre-type $(A{:}R_j) \Rightarrow S_j \to T_j$ stored in field f_j:

- the rely-set contains the field-type pairs (f, T) corresponding to all bindings $f:T$ in the record type R_j, where R_j may refer to the variable A, and
- the guarantee-set is the singleton set $\{\{(f_j,\, A \to S_j \to T_j)\}\}$, where A stands for the entire record type being checked for consistency.

The procedures Rely and Guar compute sets of field-type pairs by combining the rely-set and guarantee-set, respectively, from each premethod in R. An open record type R is consistent if $\mathsf{Guar}_A(R)$ contains all of the field-type constraints in $\mathsf{Rely}_A(R)$. The procedure Coerce converts a set of field-type pairs into a record type in the obvious way, as long as each field is mentioned in at most one pair.

Using this approach, `close normalTicker` and `close quietTicker` have type

$$\mu A.\{\ \texttt{tick}{:}IS(A)\texttt{;}\ \texttt{tock}{:}IS(A)\texttt{;}\ \texttt{loud}{:}A \to \texttt{unit} \to \texttt{bool}\ \}$$

where $IS(S) \triangleq S \to \texttt{int} \to \texttt{str}$, because the following set containment is valid:

$$\begin{aligned} &\mathsf{Guar}_A(R_0) = \{(\texttt{tick}, IS(A)), (\texttt{tock}, IS(A)), (\texttt{loud}, A \to \texttt{unit} \to \texttt{bool})\} \\ \supseteq\ &\mathsf{Rely}_A(R_0) = \{(\texttt{tick}, IS(A)), (\texttt{tock}, IS(A))\} \end{aligned}$$

Notice that the use of set containment, rather than equality, in the definition of consistency allows an open record to be used even when it stores additional premethods than those required by the recursive assumptions of others. Informally, we can think of the consistency computation as a form of record width and permutation subtyping that treats constraints on self parameters specially.

Late Typing. Once open records have been closed into ordinary mu-types, typechecking method calls can proceed as in LANGMU using standard typing rules for unfolding, record projection, and function application. A common pattern in ISOLATE is to close an open record "late" (right before a method is invoked) rather than "early" (immediately when a record is created). We introduce the following abbreviation, used on lines 17 and 18, to facilitate this pattern (note that we could use a let-binding for the `close` expression, if needed, to avoid duplicating effects):

$$e\,\#f(e') \triangleq (\texttt{unfold}\,(\texttt{close}\,e)).f\,(\texttt{close}\,e)\,(e')$$

We refer to this pattern of typechecking method invocations as "late typing," hence, the name ISOLATE to describe our extension of a standard isorecursive type system. The crucial difference between ISOLATE and LANGMU is the `close` expression, which generalizes the traditional `fold` expression while still being easy to implement. The simple set containment computation can be viewed as a way of inferring the mu-type instantiations that are required in the FSUBREC encodings of our examples. As a result, ISOLATE is able to make do with isorecursive types while still allowing premethods to be loosely mixed together.

Extension: Unbounded Polymorphism. The operation for closing records of premethods constitutes the main custom typing rule beyond LANGMU. For convenience, our formulation also includes (unbounded) parametric polymorphism. In particular, lines

```
19 val wrap :: all A,B,C,C'. (C->C') -> (A->B->C) -> (A->B->C')
20 let wrap A B C C' g f this x = g (f this x)
21
22 type OpenTickerVol =
23   { tick: (A:Tock(A)) => int  -> str
24   ; tock: (A:Tick(A)) => int  -> str
25   ; loud: (A:{})      => unit -> bool }
26
27 type TickerVol =
28   mu A. { tick, tock : A -> int -> str ; loud : A -> unit -> bool }
29
30 val louderClose :: OpenTickerVol -> TickerVol
31 let louderClose ticker =
32   let exclaim s = "! " ++ s in
33   let o1 = close normalTicker in
34   let o2 = unfold o1 in
35     fold TickerVol
36       { tick = wrap _ _ exclaim (o2.tick)
37       ; tock = wrap _ _ exclaim (o2.tock)
38       ; loud = \_. \_. true }
39
40 louderClose(quietTicker)  $ tick(2) -- "! tick ! ! tick ! ! "
41 louderClose(normalTicker) $ tick(2) -- "! tick ! tock ! tick ! tock ! "
```

Fig. 3. IsoLate Example (Part 2)

19 and 20 of Figure 3 show how to use parametric polymorphism to define a generic `wrap` function, like we saw in FSubRec.

The rest of the example in Figure 3 demonstrates a noteworthy aspect of combining late typing with message interposition. Recall that in FSubRec, premethods had to be instantiated with particular mu-types before wrapping (*cf.* the `ticker3` definition in §2.3). Using only ordinary unbounded universal quantification, however, there is no way to instantiate a pre-type in IsoLate. If trying to wrap record of mutually consistent premethods, the same result can be achieved by first closing the open object into a closed one described by a mu-type (line 33), unfolding it (line 34), and then using unbounded polymorphism to wrap its methods (lines 35 through 38). The `louderClose` function abstracts over these operations, taking an open ticker object as input and producing a closed ticker object as output. Therefore, we use `unfold` rather than `close` (signified by `$` rather than `#`) to use the resulting objects on lines 40 and 41.

Extension: Abstracting over Premethods. The previous example demonstrates how to wrap methods using unbounded polymorphism and late typing, but as with the corresponding examples in FSubRec, the approach does not help with truly wrapping premethods. If we wish to do so, we can extend IsoLate with an additional rule that allows pre-types, rather than just universally quantified types, to be instantiated with type arguments. As we will discuss, this rule offers a version of the FSubRec rule

```
42 val wrapTick :: all B,C,C'. (C -> C') ->
43                   ((A:Tick(A)) => B -> C) -> ((A:Tick(A)) => B -> C')
44 val wrapTock :: all B,C,C'. (C -> C') ->
45                   ((A:Tock(A)) => B -> C) -> ((A:Tock(A)) => B -> C')
46
47 let wrapTick B C C' g f this x = g (f[A] this x)
48 let wrapTock B C C' g f this x = g (f[A] this x)
49
50 val louder :: OpenTickerVol -> OpenTickerVol
51 let louder ticker =
52   let exclaim s = "! " ++ s in
53     { tick = wrapTock _ _ _ exclaim (ticker.tick)
54     ; tock = wrapTick _ _ _ exclaim (ticker.tock)
55     ; loud = true_ }
56
57 let (t1, t2, t3) = (louder quietTicker, louder normalTicker, louder t2)
58
59 t1 # tick(2)        -- "! tick ! ! tick ! ! "
60 t2 # tick(2)        -- "! tick ! tock ! tick ! tock ! "
61 t3 # tick(2)        -- "! ! tick ! ! tock ! ! tick ! ! tock ! ! "
```

Fig. 4. ISOLATE Example (Part 3)

for type instantiations that is limited to type variables and, hence, does not require a separate subtyping relation and equirecursive treatment types in order to reason about.

The extended system allows abstracting over premethods but requires code duplication, as in FSUBREC, for different pre-types. Notice that in the definitions of `wrapTick` and `wrapTock` (lines 42 through 48 of Figure 4), the extended system allows the premethod arguments `f` to be instantiated with the type arguments `A`. Making use of these wrapper functions allows us to define a `louder` function (lines 50 through 55) that, unlike `louderClose`, returns open objects. As a result, the expressions on lines 59 through 61 use the late typing form of method invocation.

Remarks. It is worth noting that the two extensions discussed, beyond the `close` expression, enable open object update in ISOLATE. Recall that closed objects correspond to ordinary mu-types, so traditional examples of closed object update work in ISOLATE as they do in LANGMU and the limited fragment of FSUBREC that ISOLATE supports. Open objects are not a substitute for closed objects, rather, they provide support for patterns of programming with mutually recursive sets of premethods.

As we will see next, our formulation of ISOLATE is designed to support the FSUBREC examples from § 2.3 without offering all the power of the full system. Therefore, the third row of the table in Figure 1 summarizes that ISOLATE fares as well as FSUBREC with respect to the four properties of our running examples. A prototype implementation of ISOLATE that typechecks the running example is available on the Web.[2]

[2] `https://github.com/ravichugh/late-types`

$$
\begin{array}{rrcl}
\textbf{Expressions} & e & ::= & \texttt{unit} \mid x \mid \lambda x{:}T.e \mid e_1\, e_2 \mid \Lambda A.e \mid e[T] \\
 & & \mid & \{\overline{f}=\overline{e}\} \mid e.f \mid \texttt{unfold}\, e \mid \texttt{fold}\, T\, e \\
\textit{premethod and close} & & \mid & \varsigma x{:}A<{:}T.e \mid \texttt{close}\, e \\
\textbf{Types} & R,S,T & ::= & \texttt{unit} \mid \{\,\overline{f}{:}\overline{T}\,\} \mid S \to T \mid A \mid \mu A.T \mid \forall A.\, T \\
\textit{pre-type} & & \mid & (A{:}S) \Rightarrow T \\
\textbf{Type Environments} & \Gamma & ::= & - \mid \Gamma, x{:}T \mid \Gamma, A \mid \Gamma, A<{:}T
\end{array}
$$

Fig. 5. IsoLate Syntax

3.2 Syntax and Typechecking

We now present the formal definition of IsoLate. Figure 5 defines the syntax of expressions and types, and Figure 6 defines selected typing rules; [10] provides additional definitions. We often write overbars (such as $\overline{f}{:}\overline{T}$) to denote sequences (such as $\{\,f_1{:}T_1;\ \ldots;\ f_n{:}T_n\,\}$).

Expressions. Expressions include the unit value, variables, lambdas, function application, type abstractions, type application, record literals, and record projection. The type abstraction and application forms are typical for a polymorphic lambda-calculus, where type arguments have no computational significance. Expressions also include isorecursive `fold` and `unfold` expressions that are semantically irrelevant, as usual. Unique to IsoLate are the premethod expression $\varsigma x{:}A<{:}T.e$ and the `close` e expression, which triggers consistency checking in the type system but serves no computational purpose. If we consider premethods to be another form of abstraction and `close` as a more general form of `fold`, then, in the notation from earlier sections, the syntax of IsoLate programs can be regarded as a subset of $\mathscr{L}_{\{\},\texttt{iso},\Lambda}$, the expression language of LangMu. The intended meaning of each expression form is standard. Instead of an operational semantics, we will define an elaboration semantics for IsoLate in §4.

Types. Types include the unit type, record types, function types, mu-types, universally quantified types, and type variables A, B, *etc.* Custom to IsoLate is the pre-type $(A{:}S) \Rightarrow T$ used to describe premethods, where the type A of the self parameter is bound in S (as defined by the type well-formedness rules in [10]). Type environments including bounds $A<{:}S$ for type variables that correspond to premethods and their pre-types. By convention, we use the metavariable R to describe record types.

The typechecking judgment $\Gamma \vdash e : T$ concludes that expression e has type T in an environment Γ where variables $x_1, \ldots, x_n$ have types $T_1, \ldots, T_n$, respectively. In addition to standard typechecking rules defined in [10], Figure 6 defines four custom IsoLate rules that encode a restricted form of F-bounded polymorphism.

The T-Premethod rule derives the pre-type $(A{:}S) \Rightarrow T$ for $\varsigma x{:}A<{:}S.e$ by combining the reasoning for type and value abstractions. The T-UnfoldSelf rules allows a self parameter, which is described by bounded type variables A, to be used at its upper bound T. This allows premethod self parameters to be unfolded as if they were described by mu-types (*cf.* lines 6, 7, and 8 of Figure 2). In order to facilitate abstracting over premethods, the T-PreApp rule allows a premethod to be instantiated with type

Type Checking (custom rules) $\boxed{\Gamma \vdash e : T}$

$$\frac{\Gamma, A <: S, x{:}A \vdash e : T}{\Gamma \vdash \varsigma x{:}A <: S.e : (A{:}S) \Rightarrow T}\ \text{[T-PREMETHOD]}$$

$$\frac{A <: T \in \Gamma \qquad \Gamma \vdash e : A}{\Gamma \vdash \texttt{unfold}\, e : T}\ \text{[T-UNFOLDSELF]} \qquad \frac{\Gamma \vdash e : (A{:}S) \Rightarrow T \qquad B <: S[B/A] \in \Gamma}{\Gamma \vdash e[B] : B \to T[B/A]}\ \text{[T-PREAPP]}$$

$$\frac{\Gamma \vdash e : R \qquad \mathsf{Guar}_A(R) \supseteq \mathsf{Rely}_A(R)}{\Gamma \vdash \texttt{close}\, e : \mu A.\mathsf{Coerce}(\mathsf{Guar}_A(R))}\ \text{[T-CLOSE]}$$

$$\mathsf{Guar}_A(\{\ \overline{f} : (A{:}\overline{R}) \Rightarrow \overline{S} \to \overline{T}\ \}) = \cup_i \{(f_i, A \to S_i \to T_i)\}$$
$$\mathsf{Rely}_A(\{\ \overline{f} : (A{:}\overline{R}) \Rightarrow \overline{S} \to \overline{T}\ \}) = \cup_i\, \mathsf{RelyThis}_A(R_i)$$
$$\mathsf{RelyThis}_A(\{\ \overline{f} : A \to \overline{S} \to \overline{T}\ \}) = \cup_i \{(f_i, A \to S_i \to T_i)\}$$

Fig. 6. ISOLATE Typing

variable argument B if it has the same bound S (after substitution) as the type variable A of the premethod. The effect of these two rules is to provide some of the subtypings derived by the full subtyping relation of FSUBREC.

The premises of T-CLOSE require that (1) the types of all fields f_i bound in R have the form $(A{:}R_i) \Rightarrow S_i \to T_i$ and (2) the set $\mathsf{Guar}_A(R)$ contains all of the field-type pairs in $\mathsf{Rely}_A(R)$. The guarantee-set contains pairs of the form $\{(f_i, A \to S_i \to T_i)\}$, which describes the type of the record *assuming* that all of the mutual constraints on self are satisfied. The rely-set collects all of these mutual constraints by using the helper procedure RelyThis to compute the constraints from each particular self type R_i.

To understand the mechanics of this procedure, let us consider a few examples. We define three self types

$$S_0 \triangleq \{\ \} \quad S_1(A) \triangleq \{\ \mathtt{f}{:}A \to \mathtt{unit} \to \mathtt{int}\ \} \quad S_2(A) \triangleq \{\ \mathtt{f}{:}A \to \mathtt{unit} \to \mathtt{bool}\ \}$$

that impose zero or one constraints on the receiver and three types that refer to them:

$$R_{\mathtt{f}} \triangleq \{\ \mathtt{f}{:}(A{:}S_1(A)) \Rightarrow \mathtt{unit} \to \mathtt{int}\ \}$$
$$R_{\mathtt{fg}} \triangleq \{\ \mathtt{f}{:}(A{:}S_1(A)) \Rightarrow \mathtt{unit} \to \mathtt{int};\ \mathtt{g}{:}(A{:}S_2(A)) \Rightarrow \mathtt{unit} \to \mathtt{bool}\ \}$$
$$R_{\mathtt{fh}} \triangleq \{\ \mathtt{f}{:}(A{:}S_1(A)) \Rightarrow \mathtt{unit} \to \mathtt{int};\ \mathtt{h}{:}(A{:}S_0) \Rightarrow \mathtt{unit} \to \mathtt{str}\ \}$$

The first record, $R_{\mathtt{f}}$, is consistent because its guarantee-set matches its rely-set exactly:

$$\mathsf{Guar}_A(R_{\mathtt{f}}) = \mathsf{Rely}_A(R_{\mathtt{f}}) = \{(\mathtt{f}, A \to \mathtt{unit} \to \mathtt{int})\}$$

The second, $R_{\mathtt{fg}}$, is inconsistent because the guarantee-set does not contain the rely-set:

$$\mathsf{Guar}_A(R_{\mathtt{fg}}) = \{(\mathtt{f}, A \to \mathtt{unit} \to \mathtt{int}), (\mathtt{g}, A \to \mathtt{unit} \to \mathtt{bool})\}$$
$$\not\supseteq \mathsf{Rely}_A(R_{\mathtt{fg}}) = \{(\mathtt{f}, A \to \mathtt{unit} \to \mathtt{int}), (\mathtt{f}, A \to \mathtt{unit} \to \mathtt{bool})\}$$

In particular, the second constraint in the rely-set is missing from the guarantee-set. In fact, the self types S_1 and S_2 can *never* be mutually satisfied, because they require different return types for the same field. The last record, $R_{\texttt{fh}}$, is consistent because the guarantee-set is allowed to contain fields beyond those required:

$$\begin{aligned} &\mathsf{Guar}_A(R_{\texttt{fh}}) = \{(\texttt{f}, A \to \texttt{unit} \to \texttt{int}), (\texttt{h}, A \to \texttt{unit} \to \texttt{str})\} \\ \supseteq\ &\mathsf{Rely}_A(R_{\texttt{fh}}) = \{(\texttt{f}, A \to \texttt{unit} \to \texttt{int})\} \end{aligned}$$

As noted earlier, the consistency computation resembles a form of record width and permutation subtyping implemented as set containment. In §4, we will make the connection between this approach and proper subtyping in FSUBREC.

Object Update. ISOLATE can derive, for any record type R, the judgment

$$(\varsigma \texttt{this}{:}\texttt{Self} <: R.\texttt{let}\, y = e\, \texttt{in this}) :: (\texttt{Self}{:}R) \Rightarrow \texttt{Self}$$

for a premethod that, after some well-typed expression e, returns the original self parameter. Because ISOLATE provides no analog to T-UNFOLDSELF for folding and because ISOLATE uses an isorecursive treatment of mu-types, the `this` variable is, in fact, the *only* expression that can be assigned the type `Self`. As a result, traditional (closed) object update examples require the use of mu-types, rather than pre-types, in ISOLATE.

4 Metatheory

We now show how to translate, or elaborate, ISOLATE source programs into the more expressive target language FSUBREC. Our soundness theorem establishes that well-typed programs in the source language translate to well-typed programs in the target language. The decidability of ISOLATE is evident; the primary difference beyond LANGMU is the T-CLOSE rule, which performs a straightforward computation.

4.1 The FSUBREC Calculus

We saw several examples of programming in FSUBREC in §2.3. In [10], we formally define the language and its typechecking rules. Our formulation closely follows standard presentations of equirecursive types [29] and F-bounded polymorphism [2], so we keep the discussion here brief. The language of FSUBREC expressions is standard. Notice that there are no expressions for folding and unfolding recursive types, and there is no `close` expression. The operational semantics can be found in the aforementioned references. The language of FSUBREC types replaces the isorecursive mu-types of ISOLATE with equirecursive mu-types, and adds bounded universal types $\forall A <: S.\, T$, where A is bound in S (in addition to T). To reason about bounded type variables, type environments Γ record assumptions $A <: S$. These assumptions are used by the subtyping rule S-TVAR that relates a type variable to its bound. The definitional equality of recursive types and their unfoldings is crucial for discharging the subtyping obligation in the second premise of the T-TAPP rule. As the soundness proof for the translation makes clear, the ISOLATE rules T-UNFOLDSELF and T-TAPP are restricted versions

Elaboration of Types $\boxed{[\![\Gamma]\!]}$ $\boxed{[\![T]\!]}$

$$[\![-]\!] = - \quad (1)$$
$$[\![\Gamma, x{:}T]\!] = [\![\Gamma]\!], x{:}[\![T]\!] \quad (2)$$
$$[\![\Gamma, A]\!] = [\![\Gamma]\!], A{<:}\texttt{top} \quad (3)$$
$$[\![\Gamma, A{<:}T]\!] = [\![\Gamma]\!], A{<:}[\![T]\!] \quad (4)$$

$$[\![\texttt{unit}]\!] = \texttt{unit} \quad (5)$$
$$[\![\{\ldots; f_i{:}T_i; \ldots\}]\!] = \{\ldots; f_i{:}[\![T_i]\!]; \ldots\} \quad (6)$$
$$[\![(A{:}S) \Rightarrow T]\!] = \forall A{<:}[\![S]\!].\, A \to [\![T]\!] \quad (7)$$
$$[\![S \to T]\!] = [\![S]\!] \to [\![T]\!] \quad (8)$$
$$[\![\mu A.T]\!] = \mu A.[\![T]\!] \quad (9)$$
$$[\![\forall A.\, T]\!] = \forall A{<:}\texttt{top}.\, [\![T]\!] \quad (10)$$
$$[\![A]\!] = A \quad (11)$$

Fig. 7. Translation of Environments and Types

of these two FSUBREC rules. Our soundness proof does not appeal to subtyping for function, recursive, or universal types. We include the rules S-ARROW, S-AMBER, and S-KERNEL-ALL for handling these constructs anyway, however, for reference. Part of the appeal of ISOLATE is that this extra machinery need not be implemented.

4.2 Elaboration from ISOLATE to FSUBREC

The translation from ISOLATE expressions and typing derivations to FSUBREC programs is mostly straightforward. Figure 7 defines the translation of ISOLATE types and type environments recursively, where ISOLATE pre-types are translated to FSUBREC bounded universals.

We write $D :: \Gamma \vdash e : T$ to give the name D to an ISOLATE derivation of the given judgment. In Figure 8, we define a function $[\![D]\!]$ that produces an expression e' in the target language, FSUBREC. We use this translation to define the semantics for the source language, rather than specifying an operational semantics directly. Most of the translation rules are straightforward, recursively invoking translation on subderivations. Because the expression $\texttt{unfold}\, e$ (respectively, $\texttt{fold}\, T\, e$) is intended to reduce directly to e, as usual, a derivation by the T-UNFOLD (respectively, T-FOLD) rule is translated to $[\![D_1]\!]$, the translation of the derivation of e.

The key aspects of the translation relate to the custom ISOLATE rules. Premethods correspond to polymorphic functions in the target calculus (T-PREMETHOD), so applying them to type variable arguments corresponds to type instantiation (T-PREAPP). Self parameters are described by bounded type variables in the target, so unfolding them has no computational purpose (T-UNFOLDSELF). The last noteworthy aspect is how to translate T-CLOSE derivations of expressions $\texttt{close}\, e$. Motivated by the FSUBREC encodings from §2, the idea is to create a closed record of methods by instantiating all of the (universally quantified) functions in the (translated) record with the type parameter $\mu A.\mathsf{Coerce}(\mathsf{Guar}_A([\![R]\!]))$, a mu-type that corresponds to the (converted and translated) guarantee-set of the record. Notice that every time an open record is used in a method invocation expression $e\,\#f\,(e')$, a new closed record is created in the target program. This captures the essence of late typing in ISOLATE.

Elaboration from IsoLate to FSubRec $\boxed{[\![D :: \Gamma \vdash e : T]\!] = e'}$

$$\text{[T-Unit]}\ \frac{}{[\![D :: \Gamma \vdash \texttt{unit} : \texttt{unit}]\!]} = \texttt{unit}$$

$$\text{[T-Var]}\ \frac{x{:}T \in \Gamma}{[\![D :: \Gamma \vdash x : T]\!]} = x$$

$$\text{[T-Recd]}\ \frac{\text{for } 1 \le i \le n,\ D_i :: \Gamma \vdash e_i : T_i}{[\![D :: \Gamma \vdash \{\overline{f} = \overline{e}\} : \{\overline{f} : \overline{T}\}]\!]} = \{f_1 = [\![D_1]\!];\ \cdots;\ f_n = [\![D_n]\!]\}$$

$$\text{[T-Proj]}\ \frac{D_1 :: \Gamma \vdash e : \{\ldots; f{:}T; \ldots\}}{[\![D :: \Gamma \vdash e.f : T]\!]} = [\![D_1]\!].f$$

$$\text{[T-Fun]}\ \frac{D_1 :: \Gamma, x{:}S \vdash e : T}{[\![D :: \Gamma \vdash \lambda x{:}S.e : S \to T]\!]} = \lambda x{:}S.[\![D_1]\!]$$

$$\text{[T-App]}\ \frac{D_1 :: \Gamma \vdash e_1 : S \to T \qquad D_2 :: \Gamma \vdash e_2 : S}{[\![D :: \Gamma \vdash e_1\ e_2 : T]\!]} = [\![D_1]\!]\,[\![D_2]\!]$$

$$\text{[T-TFun]}\ \frac{D_1 :: \Gamma, A \vdash e : T}{[\![D :: \Gamma \vdash \Lambda A.e : \forall A.\, T]\!]} = \Lambda A.[\![D_1]\!]$$

$$\text{[T-TApp]}\ \frac{D_1 :: \Gamma \vdash e : \forall A.\, T}{[\![D :: \Gamma \vdash e[S] : T[S/A]]\!]} = [\![D_1]\!][[\![S]\!]]$$

$$\text{[T-Fold]}\ \frac{T = \mu A.S \qquad D_1 :: \Gamma \vdash e : S[T/A]}{[\![D :: \Gamma \vdash \texttt{fold}\ T\ e : T]\!]} = [\![D_1]\!]$$

$$\text{[T-Unfold]}\ \frac{T = \mu A.S \qquad D_1 :: \Gamma \vdash e : T}{[\![D :: \Gamma \vdash \texttt{unfold}\ e : S[T/A]]\!]} = [\![D_1]\!]$$

$$\text{[T-UnfoldSelf]}\ \frac{A <: T \in \Gamma \qquad D_1 :: \Gamma \vdash e : A}{[\![D :: \Gamma \vdash \texttt{unfold}\ e : T]\!]} = [\![D_1]\!]$$

$$\text{[T-Premethod]}\ \frac{D_1 :: \Gamma, A <: S, x{:}A \vdash e : T}{[\![D :: \Gamma \vdash \varsigma x{:}A <: S.e : (A{:}S) \Rightarrow T]\!]} = \Lambda A.\lambda x{:}A.[\![D_1]\!]$$

$$\text{[T-PreApp]}\ \frac{D_1 :: \Gamma \vdash e : (A{:}S) \Rightarrow T \qquad B <: S[B/A] \in \Gamma}{[\![D :: \Gamma \vdash e[B] : B \to T[B/A]]\!]} = [\![D_1]\!][B]$$

$$\text{[T-Close]}\ \frac{\begin{array}{c} D_1 :: \Gamma \vdash e : R \qquad \mathsf{Guar}_A(R) \supseteq \mathsf{Rely}_A(R) \\ T = \mu A.\mathsf{Coerce}(\mathsf{Guar}_A(R)) \end{array}}{[\![D :: \Gamma \vdash \texttt{close}\ e : T]\!]} = \{f_1 = ([\![D_1]\!].f_1)[[\![T]\!]];\ \ldots\}$$

Fig. 8. Elaboration Semantics for IsoLate

4.3 Soundness

We now justify the correctness of our translation. Notice that because the syntactic forms of ISOLATE are so similar to those in FSUBREC, there is little value in defining an operational semantics for ISOLATE directly and then "connecting" it to FSUBREC. Instead, we use the translation to define the semantics for ISOLATE. As a result, the correctness theorem we prove needs only to state that the result of translating valid ISOLATE derivations are well-typed FSUBREC programs.

Theorem 1 (Type Soundness). *If* $D :: \Gamma \vdash e : T$, *then* $[\![\Gamma]\!] \vdash [\![D]\!] : [\![T]\!]$.

Proof. We provide the full details of the proof in [10]. Many of the cases proceed by straightforward induction. The case for T-CLOSE, which converts open records into closed records described by ordinary mu-types, is the most interesting. As discussed in §3, the key observation is that rely- and guarantee-sets can be interpreted as record types. The fact that the guarantee-set contains the rely-set can be used to argue how, with the help of definitional equality of equirecursive types, the necessary record subtypings hold via the record subtyping rule, S-RECD. As mentioned earlier, the reasoning for rules T-UNFOLDSELF and T-PREAPP appeal to T-SVAR and T-TAPP, respectively, in FSUBREC, providing a limited form of F-bounded polymorphism in ISOLATE.

5 Discussion

Our formulation of ISOLATE is a restricted version of FSUBREC that enables simple typechecking for loosely coupled premethods in a setting without explicit recursion. To conclude, we first discuss some related work that has not already been mentioned, and then we describe several ways in which future work might help to further extend the expressiveness of our system.

5.1 Related Work

Mixin and Recursive Modules. F-bounded polymorphism employs several traditional type theoretic constructs and is widely used to encode object-oriented programming patterns. Somewhat different mechanisms for combining and reusing implementations include traits, mixins, and mixin modules, which have been studied in both untyped (*e.g.* [4]) and typed (*e.g.* [14,1,19,15]) settings. Generally, these approaches distinguish expressions either as components that may get mixed in to other objects and objects which are constructed as the result of such operations. Various approaches are then used to control when it is safe to combine functionality with combinators such as sum, delete, rename, and override. Yet more expressive systems combine these approaches with full-fledged module systems and explicit recursion as found in ML (*e.g.* [33,13,32,21]).

Although all of the above approaches are more expressive than ISOLATE (which supports only sum), they rely on semantic features beyond those found in a strict lambda-calculus with records. For example, the CMS calculus [1] relies on call-by-name semantics to avoid ill-founded recursive definitions. The mixin operators of CMS can be brought to a call-by-value setting, but this requires tracking additional information

(in the form of dependency graphs) in mixin signatures [19]. In contrast, ill-founded recursion is not a concern for (call-by-value) ISOLATE; because late typing is restricted to premethods in records, the function types of these fields establish that they bind syntactic values (namely, functions). Furthermore, the target of the translation in [19] (Boudol's recursive records [3]) explicitly includes `letrec` and uses non-standard semantics for records. In contrast, the (standard) semantics of FSUBREC does not include recursive definitions. Instead, our translation relies on F-bounded quantification to tie the knot. As a result, our formulation of mixin composition can be applied to languages without explicit recursion (such as JavaScript, Python, and Ruby). In the future, we plan to investigate whether additional mixin operators can be supported for such languages.

Row Polymorphism. Row polymorphism [35,18,31] is an alternative to record subtyping where explicit type variables bind extra fields that are not explicitly named. By ensuring disjointness between named fields in a record and those described by a type variable, row polymorphism allows functions to be mixed in to different records using record concatenation operators. It is not clear how row polymorphism on its own would help, however, in a language without `fix`. We might start by writing

$$\texttt{tick} :: \forall \rho_1.\mu A.\{\,\texttt{tock}{:}A \to \texttt{int} \to \texttt{str};\ \rho_1\,\} \to \texttt{int} \to \texttt{str}$$

(and similarly for other premethods), but ρ_1 cannot be instantiated with a type that mentions A, as required for mutual recursion, because it is not in scope. The fact that row polymorphism is often used as an alternative to subtyping notwithstanding, simply adding F-bounded polymorphism to this example does not seem to help matters either.

Coeffects. Our special treatment of premethods can be viewed as function types that impose constraints on the context (in our case, self parameters) using a specification language besides that of the object type language. Several proof theories have been proposed to explicitly constrain the behavior of a program with respect to its context, for example, using modal logics [26] and coeffects [27]. These systems provide rather general mechanisms for defining and describing the notions of context, and they have been applied to dynamic binding structure, staged functional programming, liveness tracking, resource usage, and tracking cache requirements for dataflow programs. It would be interesting to see whether these approaches can be applied to our setting of objects and mutually recursive definitions.

Closed Recursion. Whereas we have focused on patterns of recursion for a language without `fix`, other researchers have studied systems with closed recursion. In particular, there have been several efforts to admit more well-behaved programs than allowed by ML-style let-polymorphism [11]. The system of polymorphic recursion [25] allows recursive calls to be instantiated nonuniformly, but the additional expressive power results in a system that is only semi-decidable. In between these two systems is Trevor Jim's proposal based on principal typings [22]. Because the notion of principal typings views the typing environment as an output of derivations, rather than an input, one can think of the environment as a set of constraints for the derived type of an expression. It could be interesting to see whether this approach can be adapted to a lambda-calculus extended with records.

5.2 Future Work and Conclusion

Our formulation is meant to emphasize that a small, syntactic variation on isorecursive mu-types can capture a set of desired usage patterns. In the future, we plan to study how additional language features — beyond the core of lambdas and records in IsoLate — interact with late typing. Important features include reference types, existential types, type operators for supporting user-defined types and interfaces [29], and record concatenation à la Wand, Remy, Mitchell, *et al.* [18].

Similar to how mutually recursive functions can be combined through self, recursive functions can also be combined through the heap. This pattern, sometimes referred to as "backpatching" or "Landin's knot," appears in imperative languages as well as module systems for functional languages [12,13]. We are studying how to adapt the idea of late typing to the setting of lambdas and references with the goal of, as in this paper, typechecking limited patterns of mutual recursion with relatively lightweight mechanisms. Overall, because languages support various kinds of (implicit) recursion through the heap and through self parameters, we believe that late typing may be useful for typechecking common programming patterns in a relatively lightweight way.

Acknowledgements. Part of this work was done while the author was at University of California, San Diego and supported by NSF grant CNS-1223850. The author wishes to thank Cédric Fournet, Ranjit Jhala, Arjun Guha, Niki Vazou, and anonymous reviewers for many helpful comments and suggestions that have improved this paper.

References

1. Ancona, D., Zucca, E.: A Calculus of Module Systems. Journal of Functional Programming (JFP) (2002)
2. Baldan, P., Ghelli, G., Raffaetà, A.: Basic Theory of F-bounded Quantification. Information and Computation (1999)
3. Boudol, G.: The Recursive Record Semantics of Objects Revisited. Journal of Functional Programming (JFP) (2004)
4. Bracha, G.: The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance. PhD thesis, University of Utah (1992)
5. Bruce, K.B., Cardelli, L., Pierce, B.C.: Comparing Object Encodings. Information and Computation (1999)
6. Canning, P., Cook, W., Hill, W., Olthoff, W., Mitchell, J.C.: F-bounded Polymorphism for Object-oriented Programming. In: Functional Programming Languages and Architecture (FPCA) (1989)
7. Cardelli, L.: A Semantics of Multiple Inheritance. In: Plotkin, G., MacQueen, D.B., Kahn, G. (eds.) Semantics of Data Types 1984. LNCS, vol. 173, pp. 51–67. Springer, Heidelberg (1984)
8. Cardelli, L.: Amber. In: Cousineau, G., Curien, P.-L., Robinet, B. (eds.) LITP 1985. LNCS, vol. 242, pp. 21–47. Springer, Heidelberg (1986)
9. Cardelli, L., Wegner, P.: On Understanding Types, Data Abstraction, and Polymorphism. Computing Surveys (1985)
10. Chugh, R.: IsoLate: A Type System for Self-Recursion, Extended Version (2015)
11. Damas, L., Milner, R.: Principal Type-Schemes for Functional Programs. In: Principles of Programming Languages (POPL) (1982)

12. Dreyer, D.: A Type System for Well-Founded Recursion. In: Principles of Programming Languages (POPL) (2004)
13. Dreyer, D.: A Type System for Recursive Modules. In: International Conference on Functional Programming (ICFP) (2007)
14. Duggan, D., Sourelis, C.: Mixin Modules. In: International Conference on Functional Programming, (ICFP) (1996)
15. Fisher, K., Reppy, J.: A Typed Calculus of Traits. In: Workshop on Foundations of Object-Oriented Programming (FOOL) (2004)
16. Ghelli, G.: Recursive Types Are Not Conservative Over Fsub. In: Bezem, M., Groote, J.F. (eds.) TLCA 1993. LNCS, vol. 664, pp. 146–162. Springer, Heidelberg (1993)
17. Greenman, B., Muehlboeck, F., Tate, R.: Getting F-Bounded Polymorphism Back in Shape. In: Programming Language Design and Implementation (PLDI) (2014)
18. Gunter, C.A., Mitchell, J.C. (eds.): Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design. MIT Press (1994)
19. Hirschowitz, T., Leroy, X.: Mixin Modules in a Call-by-Value Setting. In: ACM Transactions on Programming Languages and Systems (TOPLAS) (2005)
20. Igarashi, A., Viroli, M.: Variant Parametric Types: A Flexible Subtyping Scheme for Generics. In: ACM Transactions on Programming Languages and Systems (TOPLAS) (2006)
21. Im, H., Nakata, K., Garrigue, J., Park, S.: A Syntactic Type System for Recursive Modules. In: Object-Oriented Programming Systems, Languages, and Applications (OOPSLA) (2011)
22. Jim, T.: What Are Principal Typings and What Are They Good For? In: Principles of Programming Languages (POPL) (1996)
23. Kennedy, A.J., Pierce, B.C.: On Decidability of Nominal Subtyping with Variance, 2006. In: FOOL-WOOD (2007)
24. Leroy, X., Doligez, D., Frisch, A., Rémy, D., Vouillon, J.: OCaml System Release 4.02: Documentation and User's Manual (2014), `http://caml.inria.fr/pub/docs/manual-ocaml-4.02/`
25. Mycroft, A.: Polymorphic Type Schemes and Recursive Definitions. In: Paul, M., Robinet, B. (eds.) Programming 1984. LNCS, vol. 167, pp. 217–228. Springer, Heidelberg (1984)
26. Nanevski, A., Pfenning, F., Pientka, B.: Contextual Modal Type Theory. Transactions on Computational Logic (2008)
27. Petricek, T., Orchard, D., Mycroft Coeffects, A.: Coeffects: Unified Static Analysis of Context-Dependence. In: Fomin, F.V., Freivalds, R., Kwiatkowska, M., Peleg, D. (eds.) ICALP 2013, Part II. LNCS, vol. 7966, pp. 385–397. Springer, Heidelberg (2013)
28. Benjamin, C.: Pierce. Bounded Quantification is Undecidable. In: Principles of Programming Languages (POPL) (1992)
29. Pierce, B.C.: Types and Programming Languages. MIT Press (2002)
30. Pierce, B.C., Turner, D.N.: Simple Type-Theoretic Foundations for Object-Oriented Programming. Journal of Functional Programming (JFP) (1994)
31. Rémy, D., Vouillon, J.: Objective ML: A Simple Object-Oriented Extension of ML. In: Principles of Programming Languages (POPL) (1997)
32. Rossberg, A., Dreyer, D.: Mixin' Up the ML Module System. In: ACM Transactions on Programming Languages and Systems (TOPLAS) (2013)
33. Russo, C.: Recursive Structures for Standard ML. In: International Conference on Functional Programming (ICFP) (2001)
34. Torgersen, M., Hansen, C.P., Ernst, E.: Adding Wildcards to the Java Programming Language. Journal of Object Technology (2004)
35. Wand, M.: Complete Type Inference for Simple Objects. In: Logic in Computer Science (LICS) (1987)