Informal workshop record
of

FOOL 5

The 5th workshop on
Foundations of Object Oriented Languages

San Diego, California
17-18 January 1998

Foreword

The search for sound foundations for object-oriented languages has driven a variety of work in the theory of programming languages during the past decade, leading to a better understanding of the key concepts of object-oriented languages and to important developments in type theory, semantics, and program verification. The FOOL workshops bring together researchers to share new ideas and results.

This volume contains 9 papers presented at FOOl 5 —the 5th Workshop on Foundations of Object Oriented Languages— which was held on January 17 and 18, 1997 in San Diego California, in conjunction with the 25th Conference on Principles of Programming Languages. The papers were selected by the program committee from among a collection of 16 short five-page abstracts submitted in respond to the call for papers. The abstracts were reviewed by at least five members of the program committee who met electronically. We would like to thank all of the authors who submitted abstracts.

Workshop Committee

Workshop Chairs
Kim B. Bruce, Williams College
Benjamin C. Pierce, Indiana University

Program Chair
Didier Rémy (INRIA Rocquencourt)

Program Committee
Martín Abadi, Digital, SRC
Kathleen Fisher, AT&T Labs Research
Giorgio Ghelli, University of Pisa
Martin Hofmann, Darmstadt University of Technology
Martin Odersky, University of South Australia
Didier Rémy, INRIA Rocquencourt
David N. Turner, An Teallach Limited

Content and Schedule

Saturday 17, morning

Invited lecture: (9:00-10:00)

Objects, Local State, and Linear Polymorphism
Peter O'Hearn (Queen Mary and Westfield College)

Session 1: (10:30-12:00) Semantics interpretations of objects

Saturday 17, afternoon

Invited lecture: (13:30-14:30)

Completeness and soundness considered harmful
Greg Nelson (Digital Equipment Corporation)

Session 2: (15:00-16:30) Modular programming and virtual types

Sunday 18, morning

Invited lecture: (9:00-10:00)

Towards Diesel, a Next-Generation OO Language after Cecil
Craig Chambers (University of Washington)

Session 3: (10:30-12:00) Object extension and override

Objects and Classes in Algol-like Languages

Uday S. Reddy

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801
reddy@cs.uiuc.edu

Abstract

Many object-oriented languages used in practice descend from Algol. With this motivation, we study the theoretical issues underlying such languages via the theory of Algol-like languages. It is shown that the basic framework of this theory extends cleanly and elegantly to the concepts of objects and classes. An important idea that comes to light is that classes are abstract data types, whose theory corresponds to that of existential types. Equational and Hoare-like reasoning methods, and relational parametricity provide powerful formal tools for reasoning about Algol-like object-oriented programs.

1 Introduction

Object-oriented programming first developed in the context of Algol-like languages in the form of Simula 67 [17]. The majority of object-oriented languages used in practice claim either direct or indirect descent from Algol. Thus, it seems entirely appropriate to study the concepts of object-oriented programming in the context of Algol-like languages. This paper is an effort to formalize how objects and classes are used in Algol-like languages and to develop their theoretical underpinnings.

The formal framework we adopt is the technical notion of "Algol-like languages" defined by Reynolds [51]. The *Idealized Algol* of Reynolds is a typed lambda calculus with base types that support state-manipulation (for expressions, commands, etc.). The typed lambda calculus framework gives a "mathematical" flavor to Idealized Algol and sets it within the broader programming language research. Yet, the base types for state-manipulation make it remarkably close to practical programming languages. This combination gives us an ideal setting for studying various programming language phenomena of relevance to practical languages like C++, Modula-3, Java etc.

Reynolds also argued [50, Appendix] that object-oriented programming concepts are implicit in his Idealized Algol. The essential idea is that classes correspond to "new" operators that generate instances every time they are invoked. This obviates the need for a separate "class" concept. The idea has been echoed by others [46, 2]. In contrast, we take here the position that there is significant benefit to directly representing object-oriented concepts in the formal system instead of encoding them by other constructs. While the *effect* of classes can be obtained by their corresponding "new" operators, not all *properties* of classes are exhibited by the "new" operators. Thus, classes form a specialized form of "new" operators that are of independent interest.

In this paper, we define a language called IA^+ as an extension of Idealized Algol for object-oriented programming and study its semantics and formal properties. An important idea that comes to light is that classes are *abstract data types* whose theory corresponds to that of existential types [35]. In a sense, IA^+ is to Idealized Algol what SOL is to polymorphic lambda calculus. However, while SOL can be faithfully encoded in polymorphic lambda calculus [45], IA^+ is more constrained than Idealized Algol. The corresponding encoding does not preserve equivalences. Thus, IA^+ is a proper extension.

Related work A number of papers [19, 1, 11, 18] discuss object-oriented type systems for languages with side effects. It is not clear what contribution these type systems make to reasoning principles for programs. A related direction is that of "object encodings." Pierce and Turner [44] study the encoding of objects as abstract types, which bears some similarity to the parametricity semantics in this paper. More recent work along this line is [12]. Fisher and Mitchell [20] also relate classes to data abstraction. This work assumes a functional setting for objects, but some of the ideas deal with "state." Work on specification of stateful objects includes [5, 28, 29, 30] in addressing subtyping issues and [3, 6] in addressing self-reference issues. The major developments in the research on Algol-like languages are collected in [43]. Tennent [58] gives a gentle introduction to the concepts as of 1994.

2 The language IA^+

The language IA^+ is an extension of Idealized Algol with classes. Thus, it is a typed lambda calculus with base types corresponding to imperative programming phrases. The base types include:

- comm, the type of *commands* or state-transformers, and
- $\mathsf{exp}[\delta]$, the type of state-dependent *expressions* giving δ-typed values,

Presented at FOOL'5, 1998

- $\mathsf{val}[\delta]$, the type of phrases that directly denote δ-typed *values* (without any state-dependence).

Here, δ ranges over a collection of "data types" such as int(eger) and bool(ean) whose values are storable in variables. The "types" like $\mathsf{exp}[\delta]$ and comm are called "phrase types" to distinguish them from data types. Values of arbitrary phrase types are not storable in variables.[1]

The collection of *phrase types* (or "types," for short) is given by the following syntax:

$$\theta \quad ::= \quad \beta \mid \theta_1 \times \theta_2 \mid \theta_1 \to \theta_2 \mid \{x_1 : \theta_1, \ldots, x_n : \theta_n\} \mid \mathsf{cls}\ \theta$$

where β ranges over base types ($\mathsf{exp}[\delta]$, comm and $\mathsf{val}[\delta]$). Except for $\mathsf{cls}\ \theta$ types, the remaining type structure is that of simply typed lambda calculus with record types and subtyping. See, for instance, Mitchell [34, Ch. 10] for details. The basic subtypings include

- $\mathsf{val}[\delta] <: \mathsf{exp}[\delta]$,
- $(\mathsf{val}[\delta] \to \theta) <: (\mathsf{exp}[\delta] \to \theta)$, for a collection of types θ called "state-dependent" types, and
- the standard record subtyping ("width" as well as "depth" subtyping).

Our interpretation of subtyping is by *coercions* [34, Sec. 10.4.2]. The parameter passing mechanism of IA^+ is call-by-name (as is usual with typed lambda calculus). The second coercion above makes available Algol's notion of call-by-value. An "expression" argument can be supplied where a "value" is needed.

The type $\mathsf{cls}\ \theta$ is the type of classes that describe the behavior of θ-typed objects. An "object" is an abstraction that encapsulates some internal state represented by "fields" and provides externally visible operations called "methods." A class defines the fields and methods for a collection of objects, which are then called its "instances." The distinction between classes and instances arises because objects are stateful. (If a class is stateless, then there is no observable difference between its instances and there would be little point in making the class-instance distinction.) Classes represent the abstract (or "mathematical") concept of a behavior whereas instances represent the concrete (or "physical") realizations of the behavior.

For defining classes, we use a notation of the form:

$$\begin{array}{l}\textbf{class}\ \theta\\ \quad \textbf{fields}\ C_1\, x_1; \ldots; C_n\, x_n\\ \quad \textbf{methods}\ M\\ \quad \textbf{init}\ A\end{array}$$

The various components of the description are as follows:

- θ is a type (the type of all instances of this class), called the *signature* of the class,
- $x_1, \ldots, x_n$ are identifiers (for the fields),
- $C_1, \ldots, C_n$ are terms denoting classes (of the respective fields),
- M is a term of type θ (defining the methods of the class), and
- A is a comm-typed term (for initializing the fields).

Admittedly, this is a complex term form but it represents quite closely the term forms for classes in typical programming languages. Moreover, we will see that much of this detail has a clear type-theoretic basis.

It is noteworthy that we cannot define nontrivial classes without first having some primitive classes (needed for defining fields). We will assume a single primitive class for (mutable) variables via the constant:

$$\mathrm{Var}[\delta] : \mathsf{cls}\ \{\mathrm{get} : \mathsf{exp}[\delta],\ \mathrm{put} : \mathsf{val}[\delta] \to \mathsf{comm}\}$$

If x is an instance of $\mathrm{Var}[\delta]$ (a "variable"), then x.get is a state-dependent expression that gives the value stored in x and $x.\mathrm{put}(k)$ is a command that stores the value k in x.[2] We often use the abbreviation:

$$\mathsf{var}[\delta] \quad = \quad \{\mathrm{get} : \mathsf{exp}[\delta],\ \mathrm{put} : \mathsf{val}[\delta] \to \mathsf{comm}\}$$

for the signature type of variables. We assume the subtypings:

$$\begin{array}{l}\mathsf{var}[\delta] <: \mathsf{exp}[\delta]\\ \mathsf{var}[\delta] <: \mathsf{val}[\delta] \to \mathsf{comm}\end{array}$$

whose coercion interpretations are the corresponding field selections.

Note that the type $\mathsf{var}[\delta]$ is different from the class $\mathrm{Var}[\delta]$. Values of type $\mathsf{var}[\delta]$ need not be, in general, instances of $\mathrm{Var}[\delta]$. For instance, the following (trivial) class has instances of type $\mathsf{var}[\mathsf{int}]$:

$$\begin{array}{l}\mathrm{Trivial} = \textbf{class}\\ \qquad\qquad \mathsf{var}[\mathsf{int}]\\ \qquad \textbf{fields}\\ \qquad \textbf{methods}\ \{\mathrm{get} = 0,\\ \qquad\qquad\qquad \mathrm{put} = \lambda \mathrm{k}.\ \mathsf{skip}\}\\ \qquad \textbf{init}\ \mathsf{skip}\end{array}$$

Instances of this class always give 0 for the get message and do nothing in response to a put message. Yet they have the type $\mathsf{var}[\mathsf{int}]$. In essence, the type of an object merely gives its signature (the types of its methods), whereas its class defines its behavior. A tighter integration of classes and types would certainly be desirable. We return to this issue in Sec. 4.1.

As an example of a nontrivial class, consider the following class of counter objects:

$$\begin{array}{l}\mathrm{Counter} = \textbf{class}\\ \qquad\qquad \{\mathrm{inc}{:}\ \mathsf{comm},\ \mathrm{val}{:}\ \mathsf{exp}[\mathsf{int}]\}\\ \qquad \textbf{fields}\\ \qquad\qquad \mathrm{Var}[\mathsf{int}]\ \mathrm{cnt}\\ \qquad \textbf{methods}\\ \qquad\qquad \{\mathrm{inc} = (\mathrm{cnt.put} := \mathrm{cnt.get} + 1),\\ \qquad\qquad\ \ \mathrm{val} = \mathrm{cnt.get}\ \}\\ \qquad \textbf{init}\\ \qquad\qquad \mathrm{cnt.put}\ 0\end{array}$$

A counter has a state variable for keeping a count; the *inc* method increments the count and the *val* method returns the count. (The definition of the *inc* method could have also been written as cnt := cnt + 1 using the subtypings of $\mathsf{var}[\delta]$. We use explicit coercions for clarity.)

[1] It is possible to postulate a data type of references (or pointers) $\mathsf{ref}\ \theta$, for every phrase type θ, whose values are storable in variables. This obtains the essential expressiveness that the object-oriented programmer desires. Unfortunately, our theoretical understanding of references is not well-developed. So, we omit them from the main presentation and mention issues relating to them in Sec. 4.3.

[2] We assume that all new variables come initialized to some specific initial value init_δ. It is also possible to use a modified primitive $\mathrm{Var}[\delta]{:}\ \mathsf{exp}[\delta] \to \mathsf{cls}\ \mathsf{var}[\delta]$ that allows explicit initialization via a parameter.

One would want a variety of combinators for classes. The following "product" combinator for making pairs of objects is an essential primitive:

$$* : \mathsf{cls}\ \theta_1 \times\ \ \mathsf{cls}\ \theta_2 \to \mathsf{cls}\ (\theta_1 \times \theta_2)$$

An instance of a class $C_1 * C_2$ is a pair consisting of an instance of C_1 and an instance of C_2. Other useful combinators abound. For instance, the following combinator is motivated by the work on "fudgets" [14]:

$$\mathtt{<>} : (\theta_1 \to \mathsf{cls}\ \theta_2) \times (\theta_2 \to \mathsf{cls}\ \theta_1) \to \mathsf{cls}\ (\theta_1 \times \theta_2)$$

An instance of $F_1 \mathtt{<>} F_2$ is a pair (a, b) where a is an instance of $F_1(b)$ and b an instance of $F_1(a)$. The two objects are thus interlinked at creation time using mutual recursion. Common data structures in programming languages such as arrays and records also give rise to class combinators. The array data structure can be regarded as a combinator of type:

$$\mathrm{array} : \mathsf{cls}\ \theta \to \mathsf{val[int]} \to \mathsf{cls}\ (\mathsf{val[int]} \to \theta)$$

so that (array C n) is equivalent to an n-fold product $C * \cdots * C$, viewed as a (partial) function from integers to C-objects. The record construction

$$\mathbf{record}\ C_1\, x_1; \ldots ; C_n\, x_n\ \mathbf{end}$$

is essentially like $C_1 * \cdots * C_n$ except that its instances are records instead of tuples.

For creating instances of classes, we use the notation:

$$\mathbf{new}\ C$$

which is a value of type $(\theta \to \mathsf{comm}) \to \mathsf{comm}$ where θ is the signature of class C. For example,

new Counter λa. B

creates an instance of *Counter*, binds it to a and executes the command B. The scope of a extends as far to the right as possible, often delimited by parentheses or **begin-end** brackets.

The type of **new** C illustrates how the "physical" nature of objects is reconciled with the "mathematical" character of Algol. If **new** C were to be regarded as a value of type θ then the mathematical nature of Algol would prohibit stateful objects entirely. For example, a construction of the form

let a = **new** Counter
in a.inc; print a.val

would be useless because it would be equivalent, by β-reduction, to:

(**new** Counter).inc; print (**new** Counter).val

thereby implying that every use of a gives a new counter and no state is propagated. The higher order type of **new** C gives rise to no such problems. This insight is due to Reynolds [51] and has been used in several other languages [37, 56].

2.1 The formal system

We assume a standard treatment for the typed lambda calculus aspects of IA$^+$. The type rules for cls types are shown in Fig. 1. Note that we have one rule for the introduction of cls types and one for elimination. We show a single field in a class term for simplicity. This is obviously not a limitation because the $*$ combinator of classes can be used to instantiate multiple classes. It is significant that the initialization command is restricted to acting on the field x. We do not allow it to alter arbitrary non-local objects. The **methods** term M, on the other hand, can act on non-local objects. This is useful, for instance, to obtain the effect of "static" fields in languages like C++ and Java. If a class term does not have any free identifiers, we call it a "constant class."

$$\frac{\Gamma \rhd C : \mathsf{cls}\,\tau \quad \Gamma, x{:}\tau \rhd M : \theta \quad x{:}\tau \rhd A : \mathsf{comm}}{\Gamma \rhd (\mathbf{class}\ \theta\ \mathbf{fields}\ C\ x\ \mathbf{methods}\ M\ \mathbf{init}\ A) : \mathsf{cls}\,\theta}\ \mathsf{cls}\ \text{Intro}$$

$$\frac{\Gamma \rhd C : \mathsf{cls}\,\theta}{\Gamma \rhd \mathbf{new}\ C : (\theta \to \mathsf{comm}) \to \mathsf{comm}}\ \mathsf{cls}\ \text{Elim}$$

Figure 1: Type rules for cls types

skip	:	comm
$\_\,;\,\_$	:	$\mathsf{comm} \times \mathsf{comm} \to \mathsf{comm}$
$\mathbf{letval}_{\delta,\beta}$	:	$\mathsf{exp}[\delta] \to (\mathsf{val}[\delta] \to \beta) \to \beta$
$\mathbf{if}_\theta$	:	$\mathsf{val[bool]} \to \theta \to \theta \to \theta$
$\mathrm{Var}[\delta]$	:	$\mathsf{cls}\ \mathsf{var}[\delta]$
$\_ *_{\theta_1,\theta_2} \_$	:	$\mathsf{cls}\ \theta_1 \times \mathsf{cls}\ \theta_2 \to \mathsf{cls}\ (\theta_1 \times \theta_2)$
		(where $\beta = \mathsf{exp}[\delta']$ or comm)

Figure 2: Essential constants of IA$^+$

The restriction that the initialization command should have no free identifiers other than x is motivated by reasoning considerations. Programmers typically want to assume that the order of instance declarations is insignificant. If the initializations were to have global effects, the order would become significant. However, the restriction as stated in the rule is too stringent. One would want the initialization command to be able to at least *read* global variables. In Appendix A, we outline a more general type system based on the ideas of [50, 48] that allows read-only free identifiers.

The important constants of IA$^+$ are shown in Fig. 2. (The constants for expression and value types are omitted.) The constant skip denotes the do-nothing command and ";" denotes sequential composition. The **letval** operator sequences the evaluation of an expression with that of another expression or command. More precisely, $\mathbf{letval}\, e\, f$ evaluates e in the current state to obtain a value x and then evaluates $f\, x$. (Note that this would not make sense if $\mathbf{letval}\, e\, f$ were of type $\mathsf{val}[\delta']$.) The infix operator ":=" is a variant of **letval** defined by:

$$a := e \ \stackrel{\text{def}}{=}\ \mathbf{letval}\ e\ a$$

For example, the command (cnt.put := cnt.get + 1) in the definition of the Counter class involves such sequencing. The **letval** operator is extended to higher types as follows:

$$\begin{aligned}&\mathbf{letval}_{\delta,\theta_1\times\theta_2}\ e\ f\\ &\quad = (\mathbf{letval}_{\delta,\theta_1}\ e\ (\mathit{fst} \circ f), \mathbf{letval}_{\delta,\theta_2}\ e\ (\mathit{snd} \circ f))\\ &\mathbf{letval}_{\delta,\theta_1\to\theta_2}\ e\ f\\ &\quad = \lambda x{:}\,\theta_1.\ \mathbf{letval}_{\delta,\theta_2}\ e\ \lambda k.\ f\ k\ x\end{aligned}$$

Thus, all "state-dependent" types (as defined in Appendix A) have **letval** operators, and we have a coercion:

$$\lambda f.\, \lambda e.\, \mathbf{letval}\; e\; f : (\mathsf{val}[\delta] \to \theta) \to (\mathsf{exp}[\delta] \to \theta)$$

which serves to interpret the subtyping $(\mathsf{val}[\delta] \to \theta) <: (\mathsf{exp}[\delta] \to \theta)$.

The equational calculus for the typed lambda calculus part of IA$^+$ is standard. For cls type constructs, we have the following laws:

$$\begin{array}{ll}(\beta) & \mathbf{new}\;(\mathbf{class}\;\theta\;\mathbf{fields}\;C\;x\;\mathbf{methods}\;M\;\mathbf{init}\;A) \\ & \quad \equiv \lambda p.\,\mathbf{new}\;C\;\lambda x.\,A; p\;M \\ (\eta) & (\mathbf{class}\;\theta\;\mathbf{fields}\;C\;x\;\mathbf{methods}\;x\;\mathbf{init}\;\mathsf{skip}) \\ & \quad \equiv C \\ (\gamma) & \mathbf{new}\;C_1\;\lambda x.\,\mathbf{new}\;C_2\;\lambda y.\,M \\ & \quad \equiv \mathbf{new}\;C_2\;\lambda y.\,\mathbf{new}\;C_1\;\lambda x.\,M\end{array}$$

The (β) law specifies the effect of an Intro-Elim combination. The (η) law specifies the effect of an Elim-Intro combination where the "Elim" is the implicit elimination in field declarations. The (γ) law allows one to reorder **new** declarations. Note that it is important for initializations to be free from global effects for the (γ) law to hold.

The interaction of **new** declarations with various constants is axiomatized by the following equational axioms (for free identifiers $c : \mathsf{cls}\,\theta$, $a, b : \mathsf{comm}$, $f, g : \theta \to \mathsf{comm}$, $e : \mathsf{exp}[\delta]$, $h : \theta \to \mathsf{val}[\delta] \to \mathsf{comm}$ and $p : \mathsf{val}[\mathsf{bool}]$):[3]

$$\begin{array}{rcll}\mathbf{new}\;c\;\lambda x.\,\mathsf{skip} & = & \mathsf{skip} & (1) \\ \mathbf{new}\;c\;\lambda x.\,a; g(x) & = & a;\, \mathbf{new}\;c\;g & (2) \\ \mathbf{new}\;c\;\lambda x.\,g(x); b & = & \mathbf{new}\;c\;g;\, b & (3) \\ \left\{\begin{array}{l}\mathbf{new}\;c\;\lambda x. \\ \quad \mathbf{letval}\;e\;\lambda z.\,h\;x\;z\end{array}\right\} & = & \left\{\begin{array}{l}\mathbf{letval}\;e\;\lambda z. \\ \quad \mathbf{new}\;c\;\lambda x.\,h\;x\;z\end{array}\right\} & (4) \\ \mathbf{new}\;c\;\lambda x.\,\mathbf{if}\;p\;(f\;x)\;(g\;x) & = & \mathbf{if}\;p\;(\mathbf{new}\;c\;f)\;(\mathbf{new}\;c\;g) & (5)\end{array}$$

(In the presence of nontermination, the first equation must be weakened to an inequality $\mathbf{new}\;c\;\lambda x.\,\mathsf{skip} \sqsubseteq \mathsf{skip}$.) These equations state that the **new** operator commutes with all the operations of IA$^+$. Any computation that is independent of the new instance can be moved out of its scope. Notice that we can derive from the second equation, by setting $g = \lambda x.\,\mathsf{skip}$, the famous equation:

$$\mathbf{new}\;c\;\lambda x.\,a \quad = \quad a \qquad (6)$$

which has been discussed in various papers on semantics of local variables [31, 32, 40]. Compilers (implicitly) use these kinds of equations to enlarge or contract the scope of local variables and to eliminate "dead" variables. By formally introducing classes as a feature, we are able to generalize them for all classes.

In [50, Appendix], Reynolds suggests encoding classes as their corresponding "new" operators. This involves the translation:

$$\begin{array}{l}\mathsf{cls}\,\theta \;\rightsquigarrow\; (\theta \to \mathsf{comm}) \to \mathsf{comm} \\ (\mathbf{class}\;\theta\;\mathbf{fields}\;C\;x\;\mathbf{methods}\;M\;\mathbf{init}\;A) \\ \quad \rightsquigarrow\; \lambda p.\,\mathrm{new}\;C\;\lambda x.\,A; p(M) \\ \mathbf{new}\;C \;\rightsquigarrow\; C\end{array}$$

[3]Note that these axioms are equations of lambda calculus, not equational schemas. The symbols $c, a, g, \ldots$ are *free identifiers* which can never be substituted by terms that capture bound identifiers. For instance, in equation (2), a cannot be substituted by a term that has x occurring free.

For instance, the class *Counter* would be encoded as an operator newCounter : (counter $\to$ comm) $\to$ comm. Unfortunately, arbitrary functions of this type do not satisfy the axioms of **new** listed above. (This means that Reynolds's encoding does not give a fully abstract translation from IA$^+$ to Idealized Algol.) Our treatment can be seen as a formalization of the properties intrinsic to "new" operators of classes.

2.2 Specifications

An ideal framework for specifying classes in IA$^+$ is the *specification logic* of Reynolds [52]. Specification logic is a theory within (typed) first-order intuitionistic logic (and, hence, its name is somewhat a misnomer). We use the intuitionistic connectives "&", "$\Longrightarrow$", "$\forall$" and "$\exists$". The types include those of Idealized Algol and an additional base type assert for assertions (state-dependent classical logic formulas). The atomic formulas of specification logic include:

- equations, $M =_\theta N$, for θ-typed terms M and N,
- Hoare triples, $\{P\}\,A\,\{Q\}$, for command A and assertions P and Q, and
- non-interference formulas, $A \#, B$, where A and B are terms of arbitrary types.

Note that assertions form a "logic within logic." One can use classical reasoning for them even though the outer logic is intuitionistic. A non-interference formula $A \# B$ means intuitively that A and B do not access any common storage locations except in a read-only fashion. (A formal definition of the property uses a possible-world semantics [41].) We use a *symmetric* non-interference predicate (from [38]), which is somewhat easier to use than the original Reynolds's version. The proof rules for the non-interference predicate are the following:

1. $\ldots \& x_i \# y_j \& \ldots \Longrightarrow A \# B$ (where $\{x_i\}_i$ and $\{y_j\}_j$ are the free identifiers of A and B respectively).
2. $A \# B$ if both A and B are of "passive" types.
3. $A \# B$ if either A or B is of a "constant" type.

Passive types are those that give exp[δ]-typed values and constant types are those that give val[δ]-typed values. See Appendix A for further discussion. The effect of the non-interference predicate is best illustrated by the proof rule:

$$A \# B \Longrightarrow A; B = B; A$$

which states that two non-interfering commands can be freely reordered. The survey article of Tennent [58] has a detailed description of specification logic.

For handling IA$^+$, we extend specification logic with cls types and a new formula of the form:

$$\mathbf{Inst}\;C\;x.\,\phi(x)$$

where C is a class, x an identifier (bound in the formula) and $\phi(x)$ is a formula. The meaning is that all instances x of class C satisfy the formula $\phi(x)$. An example is the following specification of the variable class:

$$\begin{array}{l}\mathbf{Inst}\;\mathrm{Var}[\delta]\;\mathrm{x}. \\ \quad \forall \mathrm{p}\!: \mathsf{exp}[\delta] \to \mathsf{assert}.\; \mathrm{x} \# \mathrm{p} \Longrightarrow \\ \qquad \{\mathrm{p(k)}\}\;\mathrm{x.put\;k}\;\{\mathrm{p(x.get)}\}\end{array}$$

Queue : cls {init: comm, ins: val[int] $\rightarrow$ comm, del: comm, front: exp[int] }
Inst Queue q.
$\forall$x,y: val[int]. $\forall$g: exp[int] $\rightarrow$ comm. g # q $\Longrightarrow$

$$
\begin{array}{ll}
 & \text{q.init; q.init} = \text{q.init} \\
\& & \text{q.init; q.ins(x); q.del} = \text{q.init} \\
\& & \text{q.ins(x); q.ins(y); q.del} = \text{q.ins(x); q.del; q.ins(y)} \\
\& & \text{q.init; q.ins(x); g(q.front)} = \text{q.init; q.ins(x); g(x)} \\
\& & \text{q.ins(x); q.ins(y); g(q.front)} = \text{q.ins(x); g(q.front); q.ins(y)}
\end{array}
$$

Figure 3: Equational specification of a queue class

Inst Queue q.
$\exists$elems: list val[int] $\rightarrow$ assert.
$\forall$k: val[int]. $\forall$s: list val[int].

$$
\begin{array}{ll}
 & \{\text{true}\}\ \text{q.init}\ \{\text{elems}([\,])\} \\
\& & \{\text{elems(s)}\}\ \text{q.ins(k)}\ \{\text{elems(s@[k])}\} \\
\& & \{\text{elems(k::s)}\}\ \text{q.del}\ \{\text{elems(s)}\} \\
\& & \{\text{true}\}\ \text{skip}\ \{\text{elems(k::s)} \Rightarrow \text{q.front} = \text{k}\}
\end{array}
$$

Figure 4: Hoare-triple specification of queues

Queue =
  class queue
    fields (UnboundedArray Var[int]) a;
      Var[int] f, r
    methods
      {init = (f := 0; r := 0),
      ins = λx. (r := r + 1; a(r.get) := x),
      del = (**if** f < r **then** f := f + 1 **else** skip),
      front = a(f.get + 1).get }
    init (f := 0; r := 0)

Figure 5: An implementation of queues

Thus, the Hoare logic axiom for assignment becomes an axiom of the variable class. One can also write equational specifications for classes. For example, consider the specification of counters by:

Inst Counter x.
  $\forall$g: exp[int] $\rightarrow$ comm. x # g $\Longrightarrow$
    x.inc; g(x.val) = g(x.val + 1); x.inc

The quantified function identifier g plays the role of a "conversion" function, to convert expressions into commands. As a less trivial example, an equational specification of a Queue class is shown in Fig. 3. Its structure is similar to that of the Counter specification.

Specification logic allows the use of both equational reasoning and reasoning via Hoare-triples. The choice between them is a matter of preference, but Hoare-like reasoning is better understood and is often simpler. As illustration, we show in Fig. 4, a Hoare-triple specification of Queue. The specification asserts the existence of an *elems* predicate representing an abstraction of the internal state of the queue as list. (We are using an ML-like notation for lists.) Note that the logical facilities of specification logic allow us to specify the exitence of an abstraction function which would be implementation-dependent.

For example, Fig. 5 shows an implementation of the Queue class using "unbounded" arrays.[4] To show that it meets the Hoare-triple specification, we pick the predicate:

$$\text{elems(s)} \iff \text{f} \leq \text{r} \wedge \text{map a (f + 1 \ldots r) = s}$$

A Queue-state represents a queue with elements s iff $f \leq r$ and the list of array elements between $f+1$ and r is s. Note that the predicate incorporates both the "representation invariant" and the "representation function" in America's terminology [5]. In fact, all of America's theory for class specifications is implicit in specification logic.

[4] We are using "unbounded" arrays as an abstraction to finesse the technicalities of bounds. Clearly, both the specification and the implementation of Queue can be modified to deal with bounded queues.

Specification logic is also able to express "history properties" recommended by Liskov and Wing [30]. For example, here is a formula that states that a counter's value can only increase over time:

Inst Counter x.
  $\forall$k: val[int]. $\forall$g: counter $\rightarrow$ comm. x # g $\Longrightarrow$
    {x.val = k} g(x) {x.val $\geq$ k}

Using **Inst**-specifications, we formulate the following proof rule for **new** declarations:

$$\frac{\textbf{Inst}\ C\, x.\, \phi(x) \qquad \begin{array}{c}\left[\begin{array}{c}\phi(x)\\ \{C \# A_i \Longrightarrow x \# A_i\}_i\end{array}\right]\\ \vdots \\ \psi(g\, x)\end{array}}{\psi(\textbf{new}\ C\ g)} \tag{7}$$

where x does not occur free in any undischarged assumptions, the terms A_i and the formula $\psi(-)$. This states that, to prove a property ψ for (**new** $C\ g$), we need to prove ψ for $(g\ x)$, where x is an arbitrary instance of C, assuming the specification $\phi(x)$ and the fact that x does not interference with anything unless C interferes with it. The terms A_i can be any terms whatever but, in a typical usage of the rule, they are the free identifiers of $\psi(g x)$. These non-interference assumptions arise from the fact that x is a "new" instance.

The rule for inferring **Inst**-specifications is:

$$\frac{\textbf{Inst}\ C\, z.\, \psi(z) \qquad \begin{array}{c}\left[\begin{array}{c}\psi(z)\\ \{C \# A_i \Longrightarrow z \# A_i\}_i\end{array}\right]\\ \vdots \\ \phi(M)\end{array}}{\textbf{Inst}\ (\textbf{class}\ \theta\ \textbf{fields}\ C\ z\ \textbf{init}\ A\ \textbf{methods}\ M)\ x.\,\phi(x)} \tag{8}$$

where z does not occur free in any undischarged assumptions, the terms A_i and the formula $\phi(-)$.

Inst-specifications are not always adequate for capturing the entire behavior of class instances. Since they specify the behavior of instances in arbitrary states, they miss the specification of initial state and the final state transformations. Additional axioms involving **new**-terms are necessary to capture these aspects. For example, the Counter class satisfies the following "initialization" axiom:

$$\begin{aligned}&\textbf{new}\ \text{Counter}\ \lambda x.\ g(x.val);\ h(x)\\ &\quad = \textbf{new}\ \text{Counter}\ \lambda x.\ g(0);\ h(x)\end{aligned}$$

which specifies that the initial value of a counter is 0. The "finalization" axiom:

$$\begin{aligned}&\textbf{new}\ \text{Counter}\ \lambda x.\ g(x);\ h(x.inc)\\ &\quad = \textbf{new}\ \text{Counter}\ \lambda x.\ g(x);\ h(\mathsf{skip})\end{aligned}$$

states that any increment operations done just before deallocation are redundant.

3 Semantics

The denotational semantics of IA^+ brings out important properties of classes and objects. We consider two styles of semantics: *parametricity* semantics along the lines of [42], which highlights the data abstraction aspects of classes, and *object-based* semantics along the lines of [49], which highlights the class-instance relationship.

3.1 Parametricity semantics

As pointed by Reynolds [53], parametricity has to do fundamentally with data abstraction. Since classes incorporate data abstraction, one expects parametricity to play a role in their interpretation. We follow the presentation of [42, Sec. 2] in our discussion. In particular, we ignore recursion and curried functions. The later discussion in [42] in handling these features is immediately applicable.

A *type operator* T over a small collection of sets $\mathcal{S}$ is a pair $\langle T_{\mathrm{set}}, T_{\mathrm{rel}}\rangle$ where

- the "set part" T_{set} assigns to each set $X \in \mathcal{S}$, a set $T_{\mathrm{set}}(X)$, and
- the "relation part" T_{rel} assigns to each binary relation $R : X \leftrightarrow X'$, a relation $T_{\mathrm{rel}}(R) : T_{\mathrm{set}}(X) \leftrightarrow T_{\mathrm{set}}(X')$.

(We normally write both T_{set} and T_{rel} as simply T, using the context to disambiguate the notation.) Similarly, n-ary type operators with n type variables can be defined.

The type operators for constant types, variable types, product and function space constructors are standard. For example, for the function space constructor, we have the relation part:

$$\begin{aligned}&f\ [T_1(R) \to T_2(R)]\ f' \iff\\ &\qquad (\forall x, x'.\, x\ T_1(R)\ x' \implies f(x)\ T_2(R)\ f'(x'))\end{aligned}$$

The relation part for a constant type K is the identity relation, denoted Δ_K. We define quantified type operators for a universal quantifier $\forall$ and an existential quantifier $\exists$:

- The type operator $\forall Z.\, T(X, Z)$ represents *parametrically polymorphic functions* p with components $p_Z \in T(X, Z)$. Formally, its set part consists of all $\mathcal{S}$-indexed families $\{p_Z\}_{Z\in\mathcal{S}}$ such that, for all relations $S\colon Z \leftrightarrow Z'$, $p_Z\ T(\Delta_X, S)\ p_{Z'}$. Its relation part, which can be written as $\forall S.\, T(R, S)$ for any $R\colon X \leftrightarrow X'$, is defined by
$$\begin{aligned}&\{p_Z\}_{Z\in\mathcal{S}}\ \forall S.\, T(R,S)\ \{p'_Z\}_{Z\in\mathcal{S}}\\ &\iff \forall Z, Z' \in \mathcal{S}.\, \forall S\colon Z \leftrightarrow Z'.\ p_Z\ T(R,S)\ p'_{Z'}\end{aligned}$$
- The operator $\exists Z.\, T(X, Z)$ represents *data abstractions* that implement an abstract type Z with operations of type $T(X, Z)$. To define it formally, consider "implementation" pairs of the form $\langle Z, p\rangle$ where $Z \in \mathcal{S}$ and $p \in T(X, Z)$. Two such implementations are said to be *similar*, $\langle Z, p\rangle \sim \langle Z', p'\rangle$, if there exists a relation $S\colon Z \leftrightarrow Z'$ such that
$$p\ T(\Delta_X, S)\ p'$$
(Any such relation S is termed a *simulation.*) The set part $\exists Z.\, T(X, Z)$ consists of equivalence classes of implementations under the equivalence relation $\sim^*$. Write the equivalence class of $\langle Z, p\rangle$ as $\langle\!|Z, p|\!\rangle$. The relation part $\exists S.\, T(R, S)$ for any relation $R\colon X \leftrightarrow X'$ is the least relation such that
$$\langle\!|Z,p|\!\rangle\ \exists S.\, T(R,S)\ \langle\!|Z',p'|\!\rangle \Longleftarrow \exists S\colon Z \leftrightarrow Z'.\ p\ T(R,S)\ p'$$

The basic reference for parametricity is Reynolds [53], while Plotkin and Abadi [45] define a logic for reasoning about parametricity. The notion of existential quantification is from [35], but the parametricity semantics is not mentioned there. The idea of simulation relations for abstract type implementations dates back to Milner [33] and appears in various sources including [9, 27, 25, 36, 54].

The types θ of IA^+ are interpreted as type operators $[\![\theta]\!]$ in the above sense. The parameters for the type operators are state sets. Typically they capture the states involved in the representation of objects. The relation parts of the operators specify how two values of type θ are related under change of representation. Here is the interpretation:

$$\begin{aligned}[\![\mathsf{exp}[\delta]]\!](Q) &= Q \to [\![\delta]\!]\\ [\![\mathsf{comm}]\!](Q) &= Q \to Q\\ [\![\mathsf{val}[\delta]]\!](Q) &= [\![\delta]\!]\\ [\![\theta_1 \times \theta_2]\!](Q) &= [\![\theta_1]\!](Q) \times [\![\theta_2]\!](Q)\\ [\![\theta \to \beta]\!](Q) &= \forall Z.\, [\![\theta]\!](Q \times Z) \to [\![\beta]\!](Q \times Z)\\ [\![\mathsf{cls}\ \theta]\!](Q) &= \exists Z.\, [\![\theta]\!](Q \times Z) \times Z\end{aligned}$$

Note that the meaning of a class is a data abstraction. It involves a state set Z for the internal state of the instances, a component of type $[\![\theta]\!](Q \times Z)$ for the methods of the class and a component of type Z for the initial state. Two such implementations with internal state sets Z and Z' are similar (and, hence, equivalent) if, for some relation $S\colon Z \leftrightarrow Z'$, the initial states are related by S and their methods "preserve" S according to the relation $[\![\theta]\!](\Delta_Q \times S)$.

For example, consider the following class as an alternative to *Counter*:

$$\begin{aligned}\text{Counter2} = {} & \textbf{class}\{\text{inc: }\mathsf{comm},\ \text{val: }\mathsf{exp}[\mathsf{int}]\}\\ & \quad \textbf{fields}\,\text{Var}[\text{int}]\ \text{st}\\ & \quad \textbf{methods}\\ & \qquad \{\text{inc} = (\text{st.put} := \text{st.get} - 1),\\ & \qquad\ \ \text{val} = -\text{st.get}\ \}\\ & \quad \textbf{init}\,\text{st.put}\ 0\end{aligned}$$

The meanings of *Counter* and *Counter2* can be calculated as follows:

$$\begin{aligned}&[\![\text{Counter}]\!]_Q(\eta) =\\ &\quad \langle\!|\mathit{Int},\ (\{\text{inc} = \lambda(q,n).\,(q, n+1), \text{val} = \lambda(q,n).\,n\},\ 0)|\!\rangle\\ &[\![\text{Counter2}]\!]_Q(\eta) =\\ &\quad \langle\!|\mathit{Int},\ (\{\text{inc} = \lambda(q,n).\,(q, n-1), \text{val} = \lambda(q,n).\,-n\},\ 0)|\!\rangle\end{aligned}$$

The two implementations are similar because there is a simulation relation $S\colon \mathit{Int} \leftrightarrow \mathit{Int}$ given by

$$n\ S\ m \iff n \geq 0 \wedge m = -n \tag{9}$$

which is preserved by the two implementations. Hence, the two abstractions (equivalence classes) are *equal*: $[\![\text{Counter}]\!] = [\![\text{Counter2}]\!]$. Thus, the parametricity semantics gives an extremely useful proof principle for reasoning about equivalence of classes.

The interpretation of terms is as follows. A term M of type θ with free identifiers $x_1\colon\theta_1, \ldots, x_n\colon\theta_n$ is a parametric function

$$[\![M]\!] : \forall Q.\, [\![\{x_1\colon\theta_1, \ldots, x_n\colon\theta_n\}]\!](Q) \to [\![\theta]\!](Q)$$

We write $[\![M]\!]_Q$ for the component of $[\![M]\!]$ at Q. The semantics of Algol phrases is as in [42]. An important point to recall from that paper, Sec. 3.2, is the fact that parametricity makes available certain "expand" functions:

$$\mathit{expand}_\theta[Q, Z] : [\![\theta]\!](Q) \to [\![\theta]\!](Q \times Z)$$

For every value $v \in [\![\theta]\!](Q)$, there is a unique expanded value in $[\![\theta]\!](Q \times Z)$ that acts the "same way" as v does. We use the abbreviated notation $v \uparrow_Q^{Q\times Z}$ to denote $\mathit{expand}_\theta[Q, Z](v)$. For example, if $\theta = \mathsf{comm}$ and $v \in [\![\mathsf{comm}]\!](Q)$, the expanded command

$$v \uparrow_Q^{Q\times Z} = \lambda(q, z).\, (v(q), z)$$

leaves the Z component unchanged. These expand functions play a crucial role in interpreting instance declarations and inheritance. They also have significance in interpreting constants. A "constant value" in $[\![\theta]\!](Q)$ is a value of the form $v \uparrow_{\mathbf{1}}^{Q}$ obtained by expanding a value in the unit state set. So, we only need to specify the interpretation of a constant in the unit state set.

The semantics of class constructs is as follows:

$$\begin{array}{l} [\![\mathbf{class}\ \theta\ \mathbf{fields}\ C\ x\ \mathbf{methods}\ M\ \mathbf{init}\ A]\!]_Q(\eta) = \\ \quad \langle\!| Z,\ ([\![M]\!]_{Q\times Z}(\eta'[x \to m']),\ [\![A]\!]_Z[x \mapsto m'](z')) |\!\rangle \\ \qquad \text{where } \langle\!| Z,\ (m', z') |\!\rangle = [\![C]\!]_Q(\eta) \text{ and } \eta' = \eta \uparrow_Q^{Q\times Z} \end{array}$$

$$\begin{array}{l} [\![\mathbf{new}\ C\ P]\!]_Q(\eta) = \\ \quad \lambda q.\, \mathit{fst}(p_Z(m)(q, z)) \\ \qquad \text{where } \langle\!| Z, (m, z) |\!\rangle = [\![C]\!]_Q(\eta) \text{ and } p = [\![P]\!]_Q(\eta) \end{array}$$

A class definition builds an abstract type as illustrated with *Counter* above. The **new** operator "opens" the abstract type and passes to the client procedure P the representation and the method suite of the class. Thus, an "instance" is created. Note that, in the normal case where P is an abstraction $\lambda x.\, M$, its meaning is $\Lambda Z.\, \lambda m\colon [\![\theta]\!]_{Q\times Z}(\eta \uparrow_Q^{Q\times Z} [x \to m]$. So, the body term M will now use the expanded state set $Q \times Z$. Every time the class C is instantiated, a new Z component is added to the state set in this fashion. Thus, every "opening" of the abstract type gives rise to a new instance with its own state component that does not interfere with the others.

In comparing this operation with the object encoding proposed by Pierce, Turner and others [44, 12], we note that they treat *objects* as abstract types whereas we treat *classes* as abstract types. Thus, some of the bureaucratic opening-closing code that appears in their model is finessed here. Message send in our model is simply the field selection of a record. Nevertheless, the idea of abstract types appears in both the models, and the implications of this commonality should be explored further.

The class constants have the following interpretation:

$$\begin{array}{l} [\![\mathrm{Var}[\delta]]\!]_{\mathbf{1}} = \\ \quad \langle\!| [\![\delta]\!],\ (\{\mathrm{get} = \lambda d.\, d, \\ \qquad\qquad \mathrm{put} = \Lambda X.\, \lambda n.\, \lambda(d, x).\, (n, x)\}, \\ \qquad\qquad \mathbf{init}_\delta) |\!\rangle \end{array}$$

$$\begin{array}{l} [\![*]\!]_{\mathbf{1}}[Q](c_1, c_2) = \\ \quad \langle\!| Z_1 \times Z_2,\ ((m_1', m_2'), (z_1, z_2)) |\!\rangle \\ \quad \text{where} \quad \langle\!| Z_1,\ (m_1, z_1) |\!\rangle = c_1 \\ \qquad\qquad\ \ \langle\!| Z_2,\ (m_2, z_2) |\!\rangle = c_2 \\ \qquad\qquad\ \ m_1' = m_1 \uparrow_{Q\times Z_1}^{Q\times Z_1 \times Z_2} \\ \qquad\qquad\ \ m_2' = m_2 \uparrow_{Q\times Z_2}^{Q\times Z_1 \times Z_2} \end{array}$$

The $\mathrm{Var}[\delta]$ class denotes a state set $[\![\delta]\!]$ with *get* and *put* operations on it. The $*$ operator combines two classes by joining their state sets. The method suites of the individual classes are expanded to operate on the combined state set.

Theorem 1 *The parametricity model satisfies all the equivalences and axioms of Sec. 2.1.*

The plain parametricity semantics described above does not handle the equality relation in a general fashion. In implementing data abstractions, it is normal to allow the same abstract value to be represented by multiple concrete representations. In our context, this means that the equality relation for abstract states is, in general, not the same as the equality relation for concrete states. It corresponds to a partial equivalence relation (PER) for concrete states [24].

For example, in the Queue implementation of Fig. 5, an empty queue is represented by *any* state in which f and r are equal. The second axiom of the equational specification (Fig. 3) does not hold in this implementation. (The left hand side gives a state with f = r = 1 whereas the right hand side gives a state with f = r = 0.)

This can be remedied by modifying the parametricity semantics to a parametric PER semantics, where each type carries its own notion of equality.[5] More formally, A "type" in the new setting (called a PER-*type*) is a pair $X = \langle X, E_X \rangle$ where X is a set and E_X is a PER over X representing the notion of equality for X. All the above ideas can be modified to work with PER-types. (See Appendix B.)

The PER semantics influences reasoning about programs as follows. Suppose we obtain a package $\langle\langle Z, \Delta_Z\rangle, p\rangle \in \exists Z.\, T(X, Z)$ as the meaning of a class. If p preserves some PER E_Z in the sense that $p\ T(E_X, E_Z)\ p$ then we have

$$\langle\langle Z, \Delta_Z\rangle, p\rangle \sim \langle\langle Z, E_Z\rangle, p\rangle$$

Thus, we are at liberty to make up any PER E_Z that is preserved by p and use it as the equality relation for the representation.

For example, for the Queue class of Fig. 5, the state set Z consists of triples $\langle a, f, r\rangle$ where $a\colon \mathit{Int} \to \mathit{Int}$ and $f, r \in \mathit{Int}$. We pick the equivalence relation E_Z given by:

$$\begin{array}{l} \langle a, f, r\rangle\ E_Z\ \langle a', f', r'\rangle \iff \\ \qquad f \le r \wedge f' \le r' \\ \qquad \wedge\ \mathrm{map}\ a\ (f+1 \ldots r) = \mathrm{map}\ a\ (f'+1 \ldots r') \end{array}$$

to represent the intuition that only the portion of the array between $f+1$ and r contains meaningful values. In verifying the axioms of queues, we interpret $=_{\mathsf{comm}}$ as the PER for

[5] It does not seem possible to obtain the information of this semantics from the plain parametricity semantics because quantified type operators do not map PER's to PER's in general.

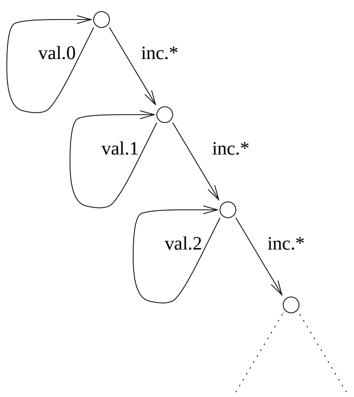

Figure 6: Trace set of a counter object

$[\![\mathsf{comm}]\!](Q \times Z)$, viz., $[E_Q \times E_Z \to E_Q \times E_Z]$. ($E_Q$ is some PER for Q respected by the other variables like g.) Here is the verification of the problematic second axiom. The two sides of the equation denote the respective state transformations:

$$\lambda(q, \langle a, f, r\rangle).\,(q,\ \langle a[1 \to x],\ 1,\ 1\rangle)$$
$$\lambda(q, \langle a, f, r\rangle).\,(q,\ \langle a,\ 0,\ 0\rangle)$$

It is clear that they are equivalent by the relation $[E_Q \times E_Z \to E_Q \times E_Z]$.

3.2 Object-based semantics

The object-based semantics [49, 39] (see also [4]) treats objects as state machines and describes them purely by their observable behavior. The observable behavior is given in terms of event traces whose structure is determined by the type of the object. This is similar to how processes are described in the semantics of CSP or CCS. Since no internal states appear in the denotations, proving the equivalence of two classes reduces to proving the equality of their trace sets.

Before looking at formal definitions, we consider an example. Figure 6 depicts the trace set of a counter object in its initial state. The events for this object are "inc.∗" denoting a successful completion of the *inc* method, and "val.i" denoting a completion of the *val* method with the result i (an integer). The nodes can be thought of as states and events as state transitions. Note that a *val* event does not change the state whereas an *inc* event takes the object to a state with a higher *val* value. For discussion purposes, we can label each node with an integer (which might well be the same integer given by *val*). The trace set can then be described mathematically by a recursive definition:

$$\begin{aligned}&\mathrm{CNT}(0) \text{ where}\\ &\quad \mathrm{CNT}(n) = \{\epsilon\} \cup \{\mathrm{inc.}{*}\} \cdot \mathrm{CNT}(n+1)\\ &\qquad\qquad\quad \cup \{\mathrm{val.}n\} \cdot \mathrm{CNT}(n)\end{aligned}$$

The parameter of the CNT function is the label of the state. Note that these labels can be anything we make up, but it often makes sense to use labels that correspond to states in an implementation. For instance, here is another description of the same trace set using negative integers for labels:

$$\begin{aligned}&\mathrm{CNT2}(0) \text{ where}\\ &\quad \mathrm{CNT2}(m) = \{\epsilon\} \cup \{\mathrm{inc.}{*}\} \cdot \mathrm{CNT2}(m+1)\\ &\qquad\qquad\quad \cup \{\mathrm{val.}(-m)\} \cdot \mathrm{CNT2}(m)\end{aligned}$$

This description corresponds to the class *Counter2*. While it is obvious that the two trace sets are the same, a formal proof would use the simulation relation S defined in (9). We can show by fixed point induction that

$$n\ S\ m \implies \mathrm{CNT}(n) = \mathrm{CNT2}(m)$$

and it follows that $\mathrm{CNT}(0) = \mathrm{CNT2}(0)$.

Note that in this description there is virtually no difference between classes and instances. A class determines a trace set which is then shared by all instances of the class. The specification equations of classes can be directly verified in the trace sets. For example, the equation $\mathrm{x.inc}; \mathrm{g(x.val)} = \mathrm{g(x.val+1)}; \mathrm{x.inc}$ of the *Counter* class is verified by noting that

$$\langle \mathrm{inc.}{*}, \mathrm{val.}(k+1)\rangle \in \mathrm{CNT}(n) \iff \langle \mathrm{val.}k, \mathrm{inc.}{*}\rangle \in \mathrm{CNT}(n)$$

for all states n.

The object-based semantics, described in [47, 39], makes these ideas work for Idealized Algol. For simplicity, we consider a version of Idealized Algol with "Syntactic Control of Interference", where functions are only applied to arguments that they do not interfere with.

We start with the notion of a coherent space [22], which is a simple form of event structure [59]. A *coherent space* is a pair $A = (|A|, \frown_A)$ where A is a (countable) set and $\frown_A$ is a reflexive-symmetric binary relation on $|A|$. The elements of $|A|$ are to be thought of as *events* for the objects of a particular type. The relation $\frown_A$, called the *coherence relation*, states whether two events can possibly be observed from the same object in the same state.

The *free object space* generated by A is a coherent space $A^* = (|A|^*, \frown_{A^*})$ where $|A|^*$ is the set of sequences over $|A|$ ("traces") and $\frown_{A^*}$ is defined by

$$\begin{aligned}&\langle a_1, \ldots, a_n\rangle \frown_{A^*} \langle b_1, \ldots, b_m\rangle \iff\\ &\quad \forall i = 1, \ldots, \mathit{min}(n, m).\\ &\qquad \langle a_1, \ldots, a_{i-1}\rangle = \langle b_1, \ldots, b_{i-1}\rangle \implies a_i \frown_A b_i\end{aligned}$$

This states that, after carrying out a sequence of events $\langle a_1, \ldots, a_{i-1}\rangle$, the two traces must have coherent events at position i. If $a_i = b_i$, then the same condition applies to position $i+1$. But if $a_i \neq b_i$, then the two events lead to distinct states and, so, there is no coherence condition on future events.

An *element* of a coherent space A is a pairwise coherent subset $x \subseteq |A|$. So, the elements of object spaces denote trace sets for objects. Functions appropriate for object spaces are what are called *regular maps* $f : A^* \to B^*$. It turns out that they can be described more simply in terms of *linear maps* $F : A^* \to B$. We actually define "multiple-argument linear maps" because they are needed for semantics. A *linear map* $F : A_1^*, \ldots, A_k^* \to B$ is a relation $F \subseteq (|A_1|^* \times \ldots \times |A_k|^*) \times |B|$ such that, whenever $(\vec{s}, b), (\vec{s'}, b') \in F$, we have

$$(\forall i.\ s_i \frown_{A_i} s_i') \implies b \frown_B b' \wedge (b = b' \implies \vec{s} = \vec{s'})$$

Every such linear map denotes a multiple-argument regular map $F^* : A_1^*, \ldots, A_k^* \to B^*$ given by

$$F^* = \{(\vec{s_1} \cdots \vec{s_n}, \langle b_1, \ldots, b_n\rangle) \mid (\vec{s_1}, b_1), \ldots, (\vec{s_n}, b_n) \in F\}$$

Coherent spaces for the events of various Idealized Algol types are shown in Figure 7. The trace sets for objects of type θ are elements of $[\![\theta]\!]^*$. Since we have a state-free description of objects, there is virtually no difference between

$$
\begin{array}{lcll}
|\mathsf{exp}[\delta]| & = & |\delta| & a \frown b \iff a = b \\
|\mathsf{comm}| & = & \{*\} & * \frown * \\
|A_1 \times A_2| & = & |A_1| + |A_2| & i.a \frown i'.a' \iff (i = i' \Longrightarrow a \frown_{A_i} a') \\
|\{l_i \colon A_i\}_i| & = & \Sigma_{l_i} A_i & l.a \frown l'.a' \iff (l = l' \Longrightarrow a \frown_{A_l} a') \\
|A^* \to B| & = & |A^*| \times |B| & (s, b) \frown (s', b') \iff (s \frown_{A^*} s' \Longrightarrow b \frown_B b' \wedge (b = b' \Longrightarrow s = s'))
\end{array}
$$

Figure 7: Coherent spaces of events for IA types

objects and classes. The only difference is that a class can be used repeatedly to generate new instances. So, a trace of a class is a sequence of object traces, one for each instance generated. Therefore, we define

$$[\![\mathsf{cls}\,\theta]\!] = [\![\theta]\!]^*$$

The meaning of a term $x_1 \colon \theta_1, \ldots, x_n \colon \theta_n \rhd M : \theta$ is a multiple-argument linear map

$$[\![M]\!] : [\![\theta_1]\!]^*, \ldots, [\![\theta_n]\!]^* \to [\![\theta]\!]$$

We regard a vector of traces $\vec{s} \in |[\![\theta_1]\!]|^* \times \ldots \times |[\![\theta_n]\!]|^*$ as a record $\eta \in \Pi_{x_i} |[\![\theta_i]\!]|^*$. So, the linear map $[\![M]\!]$ is a set of pairs (η, a), each of which indicates that, to produce an event a for the result, the term M carries out the event traces $\eta(x_i)$ on the objects for the free identifiers.

The interpretation of interference-controlled Algol terms is as in [49]. The interpretation of class terms is as follows:

$$
\begin{array}{l}
[\![\textbf{class}\;\theta\;\textbf{fields}\;C\;x\;\textbf{methods}\;M\;\textbf{init}\;A]\!] = \\
\quad \{(\eta_1 \cdot \eta_2 \cdot \eta_3, s) \mid \exists s_0, s_1 \in [\![\tau]\!]^*. \\
\qquad (\eta_1, s_0 s_1) \in [\![C]\!], \\
\qquad (\eta_2[x \to s_0], *) \in [\![A]\!], \\
\qquad (\eta_3[x \to s_1], s) \in [\![M]\!]^*\}
\end{array}
$$

$$
\begin{array}{l}
[\![\textbf{new}\;C\;P]\!] = \\
\quad \{(\eta_1 \oplus \eta_2, *) \mid \exists s \in [\![\theta]\!]^*. \\
\qquad (\eta_1, s) \in [\![C]\!], \\
\qquad (\eta_2, (s, *)) \in [\![P]\!]\}
\end{array}
$$

The meaning of the **class** term says that the trace set of C must have a trace $s_0 s_1$ where s_0 represents the effect of the initialization command A. If the methods term M maps the trace $s_1 \in |[\![\tau]\!]|^*$ to a trace $s \in |[\![\theta]\!]|^*$, then s is a possible trace for the new class. The meaning of **new** $C\ P$ finds a trace s supported by C such that P is ready to accept an object with this trace. Of course, C supports many traces. But, P will use at most one of these traces.

Theorem 2 *The object-based model satisfies all the equivalences and axioms of Sec. 2.1, adapted to a version of IA$^+$ with Syntactic Control of Interference.*

4 Modularity issues

In this section, we briefly touch upon the higher-level modularity issues relevant to object-oriented programming. Further work is needed in understanding these issues.

4.1 Types and classes

In most object-oriented languages, the notion of types and classes is fused into one. Such an arrangement is not feasible in IA$^+$ because classes are first-class values and their equality is not decidable. For example, the classes (*array c n*) and (*array c′ n′*) are equal only if n and n' are equal. Such comparisons are neither feasible nor desirable. However, a tighter integration of classes with types can be achieved using *opaque subtypes* as in Modula-3, also called "partially abstract" types [21]. For example, the counter class may be defined as:

> **newtype** counter <: {inc: comm, val: exp[int]}
> **reveal** counter = {inc: comm, val: exp[int]}
> **in**
> Counter = **class** counter ...

A client program only knows that *counter* is some subtype of the corresponding signature type and that *Counter* is of type cls *counter*. The class *Counter*, on the other hand, is inside the abstraction boundary of the abstract type *counter*, and regards it as being equal to the signature type.

We can specify requirements for partially abstract types. For example, the specification:

> ∀x: counter.
> ∀k: val[int]. ∀g: counter → comm. x # g ⟹
> {x.val = k} g(x) {x.val ≥ k}

states that *every* value of type counter — not just an instance of some class — is monotonically increasing. All **reveal** blocks of the type counter get a proof obligation to demonstrate that their use of the type counter satisfies the specification. For example, if we use **reveal** blocks to define classes Counter and Counter2, we have the job of showing that their instances are monotonically increasing. Note that such partially abstract types correspond to what America [5] calls "types."

4.2 Inheritance

Since IA$^+$ is a typed lambda calculus with records, most inheritance models in the literature can be adapted to it. For illustration, we show the recursive record model [13, 15, 46]. A class that uses self-reference is defined to be of type cls $(\theta \to \theta)$ instead of cls θ, so that the method suite is parameterized by "self." We have a combinator

> close: cls $(\theta \to \theta) \to$ cls θ
> close c = **class** θ **fields** c f **methods** fix f **init** skip

which converts a self-referential class c to a class whose instances are ordinary objects.

Let c be of type cls $(\theta \to \theta)$. To define a derived class of type cls$(\theta' \to \theta')$ where $\theta' = \theta \oplus \tau$ is a record type extension, we use a construction of the form:

> **class** $\theta' \to \theta'$
> **fields** c f; ...
> **methods** λself. (f self) **with**$[\tau]$ M
> **init** A

where **with** is a record-combination operator [16]. (The **with** operator is qualified by the type τ to indicate the record fields that get updated. This is needed for coherence of subtyping.)

As an example, suppose we define a variant of the Counter class that provides a *set* method in a "protected" fashion:

```
type protected_counter =
        counter ⊕ {set: val[int] → comm}
reveal counter = protected_counter
in
  Counter =
    class   counter → counter
      fields Var[int] cnt
      methods
          λself. {val = cnt.get,
                  set = cnt.put,
                  inc = (self.set := self.val + 1) }
      init cnt.put 0
```

We can then define a derived class that issues warnings whenever the counter reaches a specified limit:

```
reveal counter = protected_counter
in
  Warn_Counter lim =
    class   counter → counter
      fields Counter f
      methods
        λself. (f self) with[set:...]
             {set = λk. if k = lim then
                               print "Limit reached";
                          (f self).set k}
      init skip
```

Note that both (close Counter) and (close Warn_Counter) are of type cls counter. Their instances satisfy the specification of counter, including its history property. The *set* method does not cause a problem because it is inaccessible to clients.

The proof principle for self-referential classes is derived from fixed-point induction:

$$\begin{aligned}&\phi(\bot) \wedge (\textbf{Inst}\ c\ f.\, \forall x.\, \phi(x) \Longrightarrow \phi(f(x)))\\ &\quad \Longrightarrow \textbf{Inst}\ (\text{close}\ c)\ x.\, \phi(x)\end{aligned}$$

For example, both C = Counter and C = (Warn_Counter lim) satisfy:

```
Inst (close C) x.
  ∀k: val[int]. ∀p: exp[int] → assert.
        {p(k)} x.set k {p(x.val)}
    &   {p(x.val + 1)} x.inc {p(x.val)}
```

4.3 Dynamic Objects

Typical languages of Algol family provide dynamic storage via Hoare's [26] concept of "references" (pointers). An object created in dynamic storage is accessed through a reference, which is then treated as a data value and becomes storable in variables. Some of the modern languages, like Modula-3, treat references implicitly (assuming that every object is automatically a reference). But it seems preferable to make references explicit because the reasoning principles for them are much harder and not yet well-understood.

To provide dynamic storage in IA^+, we stipulate that, for every type θ, we have a data type $\mathsf{ref}\,\theta$. The operations for references are roughly as follows:

$$\frac{\Gamma \vdash C : \mathsf{cls}\,\theta}{\Gamma \vdash \textbf{newref}\ C : (\mathsf{val}[\mathsf{ref}\,\theta] \to \mathsf{comm}) \to \mathsf{comm}}$$

$$\frac{\Gamma \vdash M : \mathsf{val}[\mathsf{ref}\,\theta]}{\Gamma \vdash M\!\uparrow : \theta}$$

The rule for **newref** is not sound in general. Since references can be stored in variables and exported out of their scope, they should not refer to any local variables that obey the stack discipline. If and when the local variables are deallocated, these references would become "dangling references". A correct type rule for **newref** is given in Appendix A.

Our knowledge of semantics for dynamic storage is rather incomplete. While some semantic models exist [55, 56], it is not yet clear how to integrate them with the reasoning principles presented here.

5 Conclusion

Reynolds's Idealized Algol is a quintessential foundational system for Algol-like languages. By extending it with objects and classes, we hope to provide a similar foundation for object-oriented languages based on Algol. In this paper, we have shown that the standard theory of Algol, including its equational calculus, specification logic and the major semantic models, extends to the object-oriented setting. In fact, much of this has been already implicit in the Algol theory but perhaps in a form accessible only to specialists.

Among the issues we leave open for future work are a more thorough study of inheritance models, reasoning principles for references, and investigation of call-by-value Algol-like languages.

Acknowledgments It is a pleasure to acknowledge Peter O'Hearn's initial encouragement in the development of this work as well as his continued feedback. Bob Tennent, Hongseok Yang and the anonymous referees of FOOL 5 provided valuable observations that led to improvements in the presentation. Thanks to Martin Abadi for explaining the intricacies of PER semantics. This research was supported by the NSF grant CCR-96-33737.

Appendix

A Reflective type classes

In the type rules of section 2.1, the initialization command of a class was restricted to only the local fields of the class. While this restriction leads to clean reasoning principles: the (γ) law and equations (2-6), it is too restrictive to be practical. For instance, a counter class parameterized by an initial value n does not type-check under this restriction because its **init** command has free occurrences of n. A reasonable relaxation of the restriction is to allow the initialization command to read storage locations, but not to write to them. This kind of restriction is also useful in other contexts, e.g for defining "function procedures" that read global variables but do not modify them [58, 56].

The use of dynamic storage involves a similar restriction. A class used to instantiate a dynamic storage object should not have any references to local store. We define a general notion that is useful for formalizing such restrictions.

Definition 3 *A* reflective type class *is a set of type terms T such that*

1. $\tau_1, \tau_2 \in T \Longrightarrow \tau_1 \times \tau_2 \in T$
2. $\tau \in T \Longrightarrow \theta \to \tau \in T$
3. $\tau_1, \ldots, \tau_n \in T \Longrightarrow \{x_1 : \tau_1, \ldots, x_n : \tau_n\} \in T$

The terminology is motivated by the fact that these classes can be interpreted in reflective subcategories of the semantic category [48].

We define several reflective type classes based on the following intuitions. *Constant types* involve values that are state-independent; they neither read nor write storage locations. (Such values have been called by various qualifications such as "applicative" [56], "pure" [37], and "chaste" [57]). Dually, *state-dependent types* involve values that necessarily depend on the state. Values of *passive types* only read storage locations, but do not write to them (one of the senses of "const" in C++). Values of *dynamic types* access only dynamic storage via references.

We add three new type constructors Const, Pas and Dyn which identify the values with these properties even if they are of general types.

$$\theta \quad ::= \quad \ldots \mid \mathsf{Const}\,\theta \mid \mathsf{Pas}\,\theta \mid \mathsf{Dyn}\,\theta$$

A value of type $\mathsf{Const}\theta$ is a θ-typed value that has been built using only constant-typed information from the outside. So it can be regarded as a constant value.

We define the following classes as the least reflective classes satisfying the respective conditions:

1. *Constant types* include $\mathsf{val}[\delta]$ and $\mathsf{Const}\,\theta$ types.
2. *State-dependent types* include $\mathsf{exp}[\delta]$ and comm, and are closed under Const, Pas and Dyn type constructors.
3. *Passive types* include $\mathsf{val}[\delta]$, $\mathsf{exp}[\delta]$, $\mathsf{Const}\,\theta$ and $\mathsf{Pas}\,\theta$ types.
4. *Dynamic types* include $\mathsf{val}[\delta], \mathsf{Const}\,\theta$ and $\mathsf{Dyn}\,\theta$ types.

Definition 4 *If $\Gamma \vdash M : \theta'$, a free identifier $x{:}\,\theta$ in Γ is said to be T-*used *in M if every free occurrence of x is in a subterm of M with a T-type. (In particular, we say "constantly used", "passively used", and "dynamically used" for the three kinds of usages.)*

The introduction rules for Const, Pas, and Dyn are as follows:

$$\frac{\Gamma \vdash M : \theta}{\Gamma \vdash M : \mathsf{Const}\,\theta} \quad \text{if } \Gamma \text{ is constantly-used in } M \text{ and there are no occurrences of } \uparrow.$$

$$\frac{\Gamma \vdash M : \theta}{\Gamma \vdash M : \mathsf{Pas}\,\theta} \quad \text{if } \Gamma \text{ and } \uparrow \text{ are passively used in } M.$$

$$\frac{\Gamma \vdash M : \theta}{\Gamma \vdash M : \mathsf{Dyn}\,\theta} \quad \text{if } \Gamma \text{ is dynamically used in } M.$$

The dereference operator ($\uparrow$) is treated as if it were an identifier; Γ is T-used means that every identifier in Γ is T-used. For the elimination of these type constructors, we use the subtypings (for all types θ):

$$\begin{array}{ccccc} \mathsf{Const}\,\theta & <: & \mathsf{Pas}\,\theta & <: & \theta \\ \mathsf{Const}\,\theta & <: & \mathsf{Dyn}\,\theta & <: & \theta \end{array}$$

Note that any closed term can be given a type of the form $\mathsf{Const}\,\theta$. For example, the counter class of Section 2 has the type Const (cls *counter*).

Application to class definitions The type rule for classes is now modified as follows:

$$\frac{\Gamma \rhd C : \mathsf{cls}\,\tau \quad \Gamma, x{:}\,\tau \rhd M : \theta \quad \Gamma, x{:}\,\tau \rhd A : \mathsf{comm}}{\Gamma \rhd (\mathbf{class}\,\theta\ \mathbf{fields}\,C\,x\ \mathbf{methods}\,M\ \mathbf{init}\,A) : \mathsf{cls}\,\theta}$$
$$(\text{if } \Gamma \text{ is passively used in } A)$$

This allows the free identifiers Γ to be used in A, but in a read-only fashion. The parametricity interpretation of cls-type must be modified to $[\![\mathsf{cls}\,\theta]\!](Q) = \exists Z.\, [\![\theta]\!](Q \times Z) \times [Q \to Z]$. The rest of the theory remains the same, except that the equation (2) becomes conditional on non-interference:

$$c \,\#\, a \quad \Longrightarrow \quad \mathbf{new}\,c\,\lambda x.\,a; g(x) \quad = \quad a; \mathbf{new}\,c\,g$$

Application to references We use the following rule for creating references:

$$\frac{\Gamma \rhd C : \mathsf{Dyn}\,(\mathsf{cls}\,\theta)}{\Gamma \rhd \mathbf{newref}\,C : (\mathsf{val}[\mathsf{ref}\,\theta] \to \mathsf{comm}) \to \mathsf{comm}}$$

The rule ensures that the class instantiated in the dynamic store does not use any locations from the local store, so the instance will not use them either. This avoids the "dangling reference" problem.

B Semantics of specifications

In this section, we consider the issue of interpreting specifications. This raises two issues. First, the non-interference formulas in specifications require a sophisticated functor category interpretation [57, 41] whose relationship to the parametricity interpretation is not yet well-understood. It is however possible to interpret restricted versions of specifications, those in which $\forall$-quantified identifiers are restricted not to interfere with any other free identifiers. Note that the queue specification in Fig. 3 is of this form. The second issue, discussed in Section 3.1, is that the equality relation of specifications must be general enough to be refined by implementations.

To allow for equality relations to be refined in implementations, we define a parametric PER semantics for IA^+. The basic ideas are from Bainbridge et al. [7]. (See also [8].) We adapt them to a predicative polymorphic context. A PER E over a set X is a symmetric and transitive relation. (It differs from an equivalence relation in that it need not be reflexive.) The *domain* of E is defined by $x \in \mathrm{dom}(E) \iff x\,E\,x$. Note that E reduces to a (total) equivalence relation over $\mathrm{dom}(E)$. The set of equivalence classes under E is denoted $Q(E)$. See [34, Sec. 5.6] for discussion of PER's.

A "type" in the new setting (called a PER-*type*) is a pair $X = \langle X, E_X \rangle$ where X is a set and E_X is a PER over X. The PER specifies the notion of "equality" for the type. A "relation" $R{:}\, \langle X, E_X \rangle \leftrightarrow \langle X', E_{X'} \rangle$ is an ordinary relatin $R{:}\, X \leftrightarrow X'$ that satisfies $E_X; R; E_{X'} = R$ (called a *saturated relation*).

A "type operator" is a pair $\langle T_{\mathrm{per}}, T_{\mathrm{rel}} \rangle$ of mappings for per-types and saturated relations. The PER-type operators for products and function spaces are as follows:

$$\begin{array}{l} \langle X, E_X \rangle \times \langle Y, E_Y \rangle = \langle X \times Y,\, E_X \times E_Y \rangle \\ \qquad\qquad R \times S = R \times S \\ \langle X, E_X \rangle \to \langle Y, E_Y \rangle = \langle X \to Y,\, E_X \to E_Y \rangle \\ \qquad\qquad [R \to S] = [E_X \to E_Y];\ [R \to S];\ [E_X \to E_Y] \end{array}$$

Assume that $\mathcal{S}$ is a small collection of PER-types. We are interested in PER-type operators over $\mathcal{S}$. These operators

$$
\begin{array}{rcl}
Q,\eta \models M =_\theta N & \iff & [\![M]\!]_Q\eta \; E_{[\![\theta]\!](Q)} \; [\![N]\!]_Q\eta \\
Q,\eta \models \{P\}A\{P'\} & \iff & \forall q,q' \in Q.\, [\![P]\!]_Q\eta q = \mathsf{true} \wedge [\![A]\!]_Q\eta q = q' \Longrightarrow [\![P']\!]_Q\eta q' = \mathsf{true} \\
Q,\eta \models \phi \Longrightarrow \phi' & \iff & \forall Z.\, (Q \times Z,\, \eta \uparrow_Q^{Q\times Z} \models \phi) \Longrightarrow (Q \times Z,\, \eta \uparrow_Q^{Q\times Z} \models \phi') \\
Q,\eta \models \forall x{:}\,\theta.\, (\&_i\, x \,\#\, x_i) \Longrightarrow \phi & \iff & \forall Z. \forall v \in \mathrm{dom}(E_{[\![\theta]\!](Z)}).(Q \times Z,\, \eta \uparrow_Q^{Q\times Z} [x \to v \uparrow_Z^{Q\times Z}]) \models \phi \\
Q,\eta \models \exists x{:}\,\theta.\, \phi & \iff & \exists v \in [\![\theta]\!](Q).\, (Q,\, \eta[x \to v] \models \phi) \\
Q,\eta \models \mathbf{Inst}\; C\, x.\, \phi & \iff & \exists \langle Z, \langle p, z_0\rangle\rangle \sim^* [\![C]\!]_Q\eta.(Q \times Z,\, \eta \uparrow_Q^{Q\times Z} [x \to p] \models \phi)
\end{array}
$$

Figure 8: Interpretation of specifications

inherit product, sum and function space constructors from the above notions. We define type quantifiers as follows:

- The PER-type operator $\forall Z.\, T(X,Z)$ maps a PER-type X to the PER-type $\langle \prod_{Z\in\mathcal{S}} T(X,Z),\, \forall^{\mathrm{sat}} S.\, T(E_X,S)\rangle$ The set consists of families indexed by $Z \in \mathcal{S}$. The PER equates two families p and p' if for all saturated relations $S{:}\, Z \leftrightarrow Z'$, we have $p_Z\; T(E_X,S)\; p'_{Z'}$. The relation part of the operator maps a saturated relation $R{:}\, X \leftrightarrow X'$ to $\forall^{\mathrm{sat}} S.\, T(R,S)$.
- The PER-type operator $\exists Z.\, T(X,Z)$ maps a PER-type X to the PER-type $\langle \sum_{Z\in\mathcal{S}} T(X,Z),\, \sim_X^+\rangle$ where $\sim_X$ is given by

  $$\langle Z,p\rangle \sim_X \langle Z',p'\rangle \iff \exists^{\mathrm{sat}} S{:}\, Z \leftrightarrow Z'.\, p\; T(E_X,S)\; p'$$

  The relation part of the operator maps a saturated relation $R{:}\, X \leftrightarrow X'$ to $\sim_X^+; \exists^{\mathrm{sat}} S.\, T(R,S); \sim_{X'}^+$.

Comparing this to the plain parametricity semantics of Section 3.1, we note that PER's take the place of the identity relations.

Theorem 5 *Every type operator $T(X_1,\ldots,X_n)$ maps* PER*-types to* PER*-types and saturated relations $R_i{:}\, X_i \leftrightarrow X_i'$ to saturated relations $T(R_1,\ldots,R_n)$. Further, $T(E_{X_1},\ldots,E_{X_n}) = E_{T(X_1,\ldots,X_n)}$.*

The proof is similar to that in [8].

The interpretation of IA^+ is exactly the same as in plain parametricity semantics except that the type operators are now understood to be PER-type operators. The interpretation of specifications is shown in Fig. 8. A judgment of the form $Q,\eta \models \phi$ means that the formula ϕ with free identifiers Γ *holds* in the state set Q and environment $\eta \in \mathrm{dom}([\![\Gamma]\!](E_Q))$.

References

[1] Abadi, M., and Cardelli, L. An imperative object calculus. *Theory and Practice of Object Systems 1*, 3 (1996), 151–166.

[2] Abadi, M., and Cardelli, L. *A Theory of Objects.* Springer-Verlag, 1996.

[3] Abadi, M., and Leino, R. M. A logic of object-oriented programs. In *TAPSOFT '97 and CAAP/FASE*, vol. 1214 of *LNCS*. Springer-Verlag, 1997, pp. 682–696.

[4] Abramsky, S., and McCusker, G. Linearity, sharing and state. In *Algol-like Languages* [43], ch. 20.

[5] America, P. Designing an object-oriented programming language with behavioural subtyping. In *Foundations of Object-Oriented Languages*, J. W. de Bakker, W. P. de Roever, and G. Rozenberg, Eds., vol. 489 of *LNCS.* Springer-Verlag, 1990, pp. 60–90.

[6] Andersen, D. S., Pedersen, L. H., Hüttel, H., and Kleist, J. Objects, types and modal logics. In *FOOL 4* (http://www.cs.indiana.edu/hyplan/pierce/fool/, 1997), Electronic proceedings.

[7] Bainbridge, E. S., Freyd, P., Scedrov, A., and Scott, P. J. Functorial polymorphism. *Theoretical Comput. Sci. 70* (1990), 35–64.

[8] Berlucci, R., Abadi, M., and Curien, P.-L. A model for formal parametric polymorphism: A PER interpretation for System R. In *Typed Lambda Calculi and Applications - TLCA '95*, LNCS. Springer-Verlag, 1995, pp. 32–46. (Expanded version in a Manuscript, 1995).

[9] Borba, P., and Goguen, J. Refinement of concurrent object-oriented programs. In *Formal Methods and Object Technology*, S. Goldsack and S. Kent, Eds. Springer-Verlag, 1996, ch. 11.

[10] Brookes, S., Main, M., Melton, A., and Mislove, M., Eds. *Mathematical Foundations of Programming Semantics: Eleventh Annual Conference*, vol. 1 of *Electronic Notes in Theor. Comput. Sci.* Elsevier, 1995.

[11] Bruce, K. B. PolyTOIL: A type-safe polymorphic object-oriented language. In *ECOOP'95*, vol. 952 of *LNCS.* Springer-Verlag, 1995, pp. 27–51.

[12] Bruce, K. C., Cardelli, L., and Pierce, B. C. Comparing object encodings. Tutorial at FOOL 3 (Manuscript available from authors), 1996.

[13] Cardelli, L. A semantics of multiple inheritance. In *Semantics of Data Types*, G. Kahn, D. B. MacQueen, and G. Plotkin, Eds., vol. 173 of *LNCS.* Springer-Verlag, 1984, pp. 51–67.

[14] Carlsson, M., and Hallgren, T. FUDGETS: A graphical user interface in a lazy functional language. In *FPCA '93: Conf. on Functional Program. Lang. and Comput. Arch.* (1993), ACM, pp. 321–330.

[15] Cook, W. R. *A Denotational Semantics of Inheritance.* PhD thesis, Dep. of Computer Science, Brown Univ., May 1989. (Tech. Report CS-89-33).

[16] Cook, W. R., Hill, W. L., and Canning, P. S. Inheritance is not subtyping. In *ACM Symp. on Princ. of Program. Lang.* (1990), ACM, pp. 125–135. (Reprinted as Chapter 14 of [23]).

[17] Dahl, O.-J., and Nygaard, K. An Algol-based simulation language. *Comm. ACM 9*, 9 (Sept. 1966), 671–678.

[18] DiBlasio, P., and Fisher, K. A calculus for concurrent objects. In *CONCUR '96*, vol. 1119 of *LNCS.* Springer-Verlag, 1996, pp. 655–670.

[19] Eifrig, J., Smith, S., Trifonov, V., and Zwarico, A. An interpretation of typed OOP in a language with state. *J. Lisp and Symbolic Comput. 8*, 4 (1995), 357–397.

[20] Fisher, K., and Mitchell, J. C. Classes = objects + data abstraction. Tech. Rep. STAN-CS-TN-96-31, Stanford University, 1996.

[21] Fisher, K., and Mitchell, J. C. On the relationship between classes, objects and data abstraction. In *Mathematics of Program Construction, Marktoberdorf summer school*, LNCS. Springer-Verlag, 1997, p. (to appear).

[22] Girard, J.-Y., Lafont, Y., and Taylor, P. *Proofs and Types.* Cambridge Univ. Press, 1989.

[23] Gunter, C. A., and Mitchell, J. C., Eds. *Theoretical Aspects of Object-Oriented Programming.* MIT Press, 1994.

[24] Guttag, J. V., Horowitz, E., and Musser, D. R. Abstract data types and software validation. *Comm. ACM 21*, 12 (Dec. 1978), 1048–1063.

[25] He, J. F., Hoare, C. A. R., and Sanders, J. W. Data refinement refined. vol. 213 of *LNCS*. Springer-Verlag, 1986, pp. 187–196.

[26] Hoare, C. A. R. Record handling. In *Programming Languages: NATO Advanced Study Institute*, F. Genuys, Ed. Academic Press, London, 1968, pp. 291–347.

[27] Kinoshita, Y., O'Hearn, P. W., Power, A. J., Takeyama, M., and Tennent, R. D. An axiomatic approach to binary logical relations with applications to data refinement. Manuscript, Queen Mary and Westfield, London, 1997.

[28] Lano, K., and Haughton, H. Reasoning and refinement in object-oriented specification languages. In *ECOOP '97*, O. L. Madsen, Ed., vol. 615 of *LNCS*. Springer-Verlag, 1992, pp. 78–97.

[29] Leavens, G. T. Modular specification and verification of object-oriented programs. *IEEE Software* (July 1991), 72–80.

[30] Liskov, B., and Wing, J. M. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst. 16*, 6 (1994), 1811–1841.

[31] Mason, I. A., and Talcott, C. L. Axiomatizing operational equivalence in the presence of side effects. In *Proceedings, Fourth Annual Symposium on Logic in Computer Science* (June 1989), IEEE Computer Society Press, pp. 284–293.

[32] Meyer, A. R., and Sieber, K. Towards fully abstract semantics for local variables. In *Fifteenth Ann. ACM Symp. on Princ. of Program. Lang.* (1988), ACM, pp. 191–203. (Reprinted as Chapter 7 of [43]).

[33] Milner, R. An algebraic definition of simulation between programs. In *Proc. Second Intern. Joint Conf. on Artificial Intelligence* (London, 1971), The British Computer Society, pp. 481–489.

[34] Mitchell, J. C. *Foundations of Programming Languages.* MIT Press, 1997.

[35] Mitchell, J. C., and Plotkin, G. D. Abstract types have existential types. *ACM Trans. Program. Lang. Syst. 10*, 3 (1988), 470–502.

[36] Morgan, C., and Vickers, T., Eds. *On the Refinement Calculus.* Springer-Verlag, 1992.

[37] Odersky, M., Rabin, D., and Hudak, P. Call by name, assignment and the lambda calculus. In *Twentieth Ann. ACM Symp. on Princ. of Program. Lang.* (1993), ACM.

[38] O'Hearn, P. W., Power, A. J., Takeyama, M., and Tennent, R. D. Syntactic control of interference revisited. In Brookes et al. [10]. (Reprinted as Chapter 18 of [43]).

[39] O'Hearn, P. W., and Reddy, U. S. Objects, interference and Yoneda embedding. In Brookes et al. [10].

[40] O'Hearn, P. W., and Tennent, R. D. Semantics of local variables. In *Applications of Categories in Computer Science*, M. P. Fourman, P. T. Johnstone, and A. M. Pitts, Eds. Cambridge Univ. Press, 1992, pp. 217–238.

[41] O'Hearn, P. W., and Tennent, R. D. Semantical analysis of specification logic, Part 2. *Inf. Comput. 107*, 1 (1993), 25–57. (Reprinted as Chapter 14 of [43]).

[42] O'Hearn, P. W., and Tennent, R. D. Parametricity and local variables. *J. ACM 42*, 3 (1995), 658–709. (Reprinted as Chapter 16 of [43]).

[43] O'Hearn, P. W., and Tennent, R. D. *Algol-like Languages (Two volumes).* Birkhäuser, Boston, 1997.

[44] Pierce, B. C., and Turner, D. N. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming 4*, 2 (Apr. 1994), 207–247.

[45] Plotkin, G., and Abadi, M. A logic for parametric polymorphism. In *Typed Lambda Calculi and Applications - TLCA '93* (1993), LNCS, Springer-Verlag, pp. 361–375.

[46] Reddy, U. S. Objects as closures: Abstract semantics of object-oriented languages. In *ACM Symp. on LISP and Functional Program.* (July 1988), ACM, pp. 289–297.

[47] Reddy, U. S. Global state considered unnecessary: Semantics of interference-free imperative programming. In *ACM SIGPLAN Workshop on State in Program. Lang.* (June 1993), Technical Report YALEU/DCS/RR-968, pp. 120–135.

[48] Reddy, U. S. Passivity and independence. In *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science* (July 1994), IEEE Computer Society Press, pp. 342–352.

[49] Reddy, U. S. Global state considered unnecessary: An introduction to object-based semantics. *J. Lisp and Symbolic Computation 9* (1996), 7–76. (Reprinted as Chapter 19 of [43]).

[50] Reynolds, J. C. Syntactic control of interference. In *ACM Symp. on Princ. of Program. Lang.* (1978), ACM, pp. 39–46. (Reprinted as Chapter 10 of [43]).

[51] Reynolds, J. C. The essence of Algol. In *Algorithmic Languages*, J. W. de Bakker and J. C. van Vliet, Eds. North-Holland, 1981, pp. 345–372. (Reprinted as Chapter 3 of [43]).

[52] Reynolds, J. C. Idealized Algol and its specification logic. In *Tools and Notions for Program Construction*, D. Neel, Ed. Cambridge Univ. Press, 1982, pp. 121–161. (Reprinted as Chapter 6 of [43]).

[53] Reynolds, J. C. Types, abstraction and parametric polymorphism. In *Information Processing '83*, R. E. A. Mason, Ed. North-Holland, Amsterdam, 1983, pp. 513–523.

[54] Schoett, O. Behavioral correctness of data representations. *Sci. Comput. Programming 14* (1990), 43–57.

[55] Stark, I. Categorical models for local names. *Lisp and Symbolic Computation 9*, 1 (Feb. 1996), 77–107.

[56] Swarup, V., Reddy, U. S., and Ireland, E. Assignments for applicative languages. In *Algol-like Languages* [43], ch. 9, pp. 235–272.

[57] Tennent, R. D. Semantical analysis of specification logic. *Inf. Comput. 85*, 2 (1990), 135–162. (Reprinted as Chapter 13 of [43]).

[58] Tennent, R. D. Denotational semantics. In *Handbook of Logic in Computer Science*, S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, Eds., vol. 3. Oxford University Press, 1994, pp. 169–322.

[59] Winskel, G. An introduction to event structures. In *Linear Time, Branching Time and Partial Order in Logics and Models of Concurrency*, J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, Eds., vol. 354 of *LNCS*. Springer-Verlag, 1989, pp. 364–397.

Behavioural Subtyping for a Type-Theoretic Model of Objects

Erik Poll

E.Poll@ukc.ac.uk

Computing Laboratory, University of Kent, Canterbury, UK

Abstract

We present a refinement of the existential object model of Pierce and Turner [PT94]. In addition to signatures (or interfaces) as the types of objects, we also provide *classes* as the types of objects. These class types not only specify an interface, but also a particular implementation.

We show that class types can be interpreted in the standard PER model. Our main result is that the standard interpretation of subtyping in PER models – i.e. subtypes are subpers – is then *behavioural subtyping* in the sense of Leavens [Lea90].

1 Introduction

Like most descriptions of objects in typed lambda calculus or typed object calculus, the existential object model of Pierce and Turner [PT94] provides *signatures* or *interfaces* as the types of objects, and provides the usual syntactic notion of subtyping on these types. We consider a more refined notion of *class type* as the type of an object. A class type is a subtype of an interface type, specified by a particular implementation and initial state.

The existential object encoding can be carried out in $F^\omega_\leq$, but the class types we want to consider are not encodable in $F^\omega_\leq$. Therefore we will define a simple object-oriented functional language λ^{OO}, that is essentially just some syntactic sugar for the existential object encoding, but extended with a notion of class type.

The standard model of $F^\omega_\leq$ is a PER model, which interprets types as partial equivalence relations (pers), and interprets subtyping as subset inclusion between pers [BL90]. We show that class types can be interpreted in the PER model. For these interpretations of class types the subset relation turns out to be equivalent to the notion of *behavioural subtyping* as defined by Leavens [Lea90].

It is important to note that class types can be behavioural subtypes even though they give completely unrelated implementations. Although class types correspond to particular implementations, behavioural subtyping between class types only concerns the observable behaviour of these implementations.

The interpretation of class types can also be viewed from the categorical perspective used in [Rei95] and [HP95]: class types are interpreted as sub-coalgebras of final coalgebras in the PER model.

We briefly review the different notions of (sub)typing for objects, and fix the terminology used for them in this paper.

Interface Types vs Class Types

There are at least two kinds of types that can be used for objects, which we will call *interface types* and *class types.* An *interface type* just specifies an interface (or signature), i.e. it lists the methods with their input and output types. A *class type* not only specifies the signatures of the methods, but also an implementation of the methods and an initial state. An important consequence of this is that the objects of a class type can be guaranteed to have some behaviour in common. The objects of an interface type on the other hand cannot be guaranteed to have any behaviour in common.

Most object-oriented languages provide classes as types. Most type-theoretical work on OO on the other hand concerns interface types (e.g. [Car88], many of the papers in [GM94], [Bru94], [FM94], [AC96]). A notable exception is [FM97], which gives an extensive comparison of interface types and class types.

Signature Subtyping vs Behavioural Subtyping

There are at least two notions of subtyping for object types, which we will call *signature subtyping* and *behavioural subtyping.*

Signature subtyping is a purely syntactic notion, defined by the usual contra/covariant rules (e.g. [Car88]). It concerns just the interfaces of objects: a signature subtype provides (at least) all the methods of the supertype with compatible signatures. Signature subtyping prevents type errors ("message not understood") from occurring at run-time: if A is a signature subtype of B then substituting A's for B's will not cause any type errors.

Behavioural subtyping is a more semantic notion, and concerns more than just signatures. It also tries to capture the intuition that objects in the subtype "behave like" objects in the supertype. It can informally be defined as follows: A is a behavioural subtype of B iff using objects of type A in place of objects of type B does not cause any unexpected behaviour. Or, A is a behavioural subtype of B iff any property that holds for all objects of type B also holds for all objects of type A. Another formulation is known as Liskov's substitution principle [Lis88]: "A is a behavioural subtype of B iff for every object a of type A there is an object b of type B such that for all programs p that use b, the behaviour of p is unchanged when b is replaced with a".

Behavioural subtyping is clearly a useful property for reasoning about programs. Behavioural subtyping is in general not decidable, unlike signature behavioural subtyping, which can be statically enforced by a typechecker.

There are different ways to give formal definitions of behavioural subtyping.

One way to define behavioural subtyping is to require there exists a simulation relation between states of objects in the subtype and states of objects in the supertype such that for every object in the behavioural subtype there is an object in the supertype with a related state. This characterisation is the one we will use. It was introduce by Leavens in [Lea90], and is also used in [LW95] [Mau95].

Another – more common – way to define behavioural subtyping is in terms of pre- and post-conditions of methods, and require that methods in a behavioural subtype have weaker pre-conditions and stronger post-conditions than the corresponding methods in the supertypes, so that behavioural subtypes correspond to stronger specifications. This is supported to a certain extent in the programming language Eiffel [Mey88], and is widely used in the literature, e.g. [Ame89] [Lea90] [LW95] [LW94] [PH97] [AL97]. This second approach can combined with the first in order to cope with pre- and postconditions that refer to the abstract values of objects. For example, suppose objects of type $List_A$ are specified in terms of the sequences in $A*$ they represent, and objects of type Set_A in terms of the sets in $\mathcal{P}(A)$ they represent. Then a (simulation) relation $R \subseteq Set_A \times \mathcal{P}(A)$ could be used to relate these specifications. This approach is used in [Lea90]. It is also used in [Ame89][LW94], where simulation relations are restricted to functions.

The rest of this paper is organised as follows. Section 2 gives some examples to illustrate the notions mentioned above. Section 3 then gives a formal definition of λ^{OO}. Section 4 discusses the PER model for $F_{\leq}$ of [PAC94], which provides a PER model for λ^{OO} without class types. This PER model is then used in Section 5 as the basis of a PER model for λ^{OO} including the class types, and we show that subtyping between class types in this model is behavioural subtyping. We conclude in Section 6.

2 Informal Introduction to λ^{OO}

We give some simple examples to introduce λ^{OO} and the existential object model, and to explain the notions of interface types, class types, signature subtyping, and behavioural subtyping in more detail.

2.1 Objects and Interface Types

The interface type

$$Counter \quad = \quad \mathbf{Sig}\ (X)\{getcount : Nat, count : X\}$$

is the type of objects with methods *getcount* and *count*, where *getcount* returns a natural number and *count* a new object of the same type. We write $o{\leftarrow}l$ for the invocation of method l of object o. So, if $o : Counter$ then $o{\leftarrow}getcount : Nat$ and $o{\leftarrow}count : Counter$.

An object of type $Counter$ – a counter – can be constructed from a state s of some type Rep and a method table $m : Rep{\to}CounterI(Rep)$ that gives the implementation of the methods, where

$$CounterI(X) \quad = \quad \{getcount : Nat, count : X\}.$$

This object is written as **object** $\langle s, m\rangle$.

For example, we could use the type $\{x : Nat\}$ – the type of records with an x-field of type Nat – to represent the state of counter, and use the following function as method table

$$\begin{array}{rcl} m & = & \lambda s : \{x : Nat\}.\ \{get = s.x, count = \{x = s.x + 1\}\} \\ & : & \{x : Nat\}{\to}CounterI(\{x : Nat\}) \end{array}$$

E.g. **object** $\langle\{x = 5\}, m\rangle$ is the counter-object with $\{x = 5\}$ as state and m as method table.

A simple operational semantics for method invocation can be given as follows

$$\begin{array}{rcl} (\mathbf{object}\ \langle s, m\rangle){\leftarrow}getcount & = & (m\ s).getcount \\ (\mathbf{object}\ \langle s, m\rangle){\leftarrow}count & = & \mathbf{object}\ \langle(m\ s).count, m\rangle \end{array}$$

For example, if $o \equiv \mathbf{object}\langle\{x = 5\}, m\rangle$, then $o{\leftarrow}getcount = 5$ and $o{\leftarrow}count = \mathbf{object}\ \langle\{x = 6\}, m\rangle$. Note that the methods *count* and *getcount* are treated differently, because *count* returns an object of the same class and *getcount* just returns a natural number. We will call a method such as *count* a *mutator* and a method such as *getcount* an *observer*. In general, to invoke a method l we apply the method table to the state and select l component, and, if the method is mutator, we wrap up the new state with the old method table to produce an object.

Everything described so far is just syntactic sugar for the existential object encoding of [PT94]. We have written **Sig** $CounterI$ for

$$\exists Rep.Rep \times (Rep{\to}CounterI(Rep))$$

and **object** $\langle s, m\rangle$ for

$$\mathbf{pack}\ \langle Rep, (s, m)\rangle\ \mathbf{to}\ \exists Rep.Rep \times (Rep{\to}CounterI(Rep))$$

where Rep is the type of s.

Existential types model abstract types [MP88]. The existential type above hides the type Rep that used to represent the state of an object, so that the state s of an object **object** $\langle s, m\rangle$ can only be observed indirectly by invoking the methods *getcount* and *count*.

In certain models – including the PER model we will use – the existential type above is interpreted as a final coalgebra [PA93][Has94]. Then equality of objects is *bisimulation*: **object** $\langle s_1, m_1\rangle$ and **object** $\langle s_2, m_2\rangle$ of type $Counter$ are equal if their states are related by some simulation relation $\sim$, i.e. if $s_1 \sim s_2$ for some relation $\sim$ such that

$$x_1 \sim x_2 \Rightarrow \left\{ \begin{array}{c} (m_1\ x_1).getcount =_{Nat} (m_2\ x_2).getcount \\ (m_2\ x_1).count \sim (m_2\ x_2).count \end{array} \right.$$

2.2 Subtyping on Interface Types

An example of a signature subtype of $Counter$ is

$$RCounter \quad = \quad \mathbf{Sig}\, RCounterI,$$

where

$$RCounterI(X) \quad = \quad \{getcount : Nat, count : X, reset : X\}.$$

$RCounter$ is type of counters that in addition to methods $count$ and $getcount$ also have a method $count$.

The PER model does indeed interpret $RCounter$ and $Counter$ as subsets: $[\![RCounter]\!] \subseteq [\![Counter]\!]$. We can think of the partial equivalence relations $[\![RCounter]\!]$ and $[\![Counter]\!]$ as the notion of equality for counters and resetable counters, respectively. Then $[\![RCounter]\!] \subseteq [\![Counter]\!]$ means that $[\![Counter]\!]$ is a coarser notion of equality than $[\![RCounter]\!]$. This can be understood as follows. Suppose we have two objects that can not be distinguished by invoking just the methods $count$ and $getcount$, but that can be distinguished by invoking the methods $count$, $getcount$ and $reset$. These objects would then be identified by $[\![Counter]\!]$, but would not be identified by $[\![RCounter]\!]$.

The subtyping between $RCounter$ and $Counter$ is a degenerated instance of behavioural subtyping. For interface types such as $RCounter$ and $Counter$ the notions of signature subtyping and behavioural subtyping are identical, because these types do not make any behavioural guarantees.

2.3 Class Types

The class types we consider not only specify an interface, but also fix the type used for the representation, the implementation of the methods, and an initial state. An example of a class type is

$$CounterClass \quad = \quad (\mathbf{Class}\, CounterI \,\mathbf{with}\, init, m)$$

where

$$\begin{array}{rcl} init & = & \{x = 1\} \\ m & = & \lambda s{:}\{x : Nat\}.\, \{get = s.x, count = \{x = s.x + 1\}\} \end{array}$$

Intuitively, $CounterClass$ is the following inductively defined set:

- $\mathbf{object}\ \langle init, m\rangle : CounterClass$,
- if $c : CounterClass$ then $c{\leftarrow}count : CounterClass$.

i.e. $CounterClass$ can be understood as the smallest subset of $Counter$ that contains $\mathbf{object}\ \langle init, m\rangle$ and is closed under method invocation.

$CounterClass$ is like a class definition in a standard OO language, in that it introduces a type together with an implementation and initialisation for the objects of that type. In a programming language we might write something like "$new.CounterClass$" for $\mathbf{object}\ \langle init, m\rangle$.

Because the objects in $CounterClass$ all have m as their method table and all have a hidden state of type $\{x : Nat\}$ that is "reachable" from the initial state s by method invocations, they have some behaviour in common. For example,

$$o{\leftarrow}getcount \geq 1$$

and

$$(o{\leftarrow}count){\leftarrow}getcount = o{\leftarrow}getcount + 1$$

for all $o : CounterClass$.

Interpreting class types in the PER model is not a problem, and we get the expected relation between the pers interpreting $CounterClass$ and $Counter$: $[\![CounterClass]\!] \subseteq [\![Counter]\!]$. The pers $[\![CounterClass]\!]$ and $[\![Counter]\!]$ provide the same notion of equality, i.e. all $[\![CounterClass]\!]$ equivalence classes are also $[\![Counter]\!]$equivalence classes, but there are fewer $[\![CounterClass]\!]$ equivalence classes than there are $[\![Counter]\!]$equivalence classes.

2.4 Subtyping on Class Types

Assume there is a type $natlist$ of lists of natural numbers, with the usual constructors nil and $cons$. The class type below uses a state of type $\{y : natlist\}$ to represent the state of a resetable counter:

$$RCounterClass = (\mathbf{Class}\, RCounterI \,\mathbf{with}\, init_R, m_R)$$

where

$$\begin{array}{rcl} init_R & = & \{y = cons\ 0\ (cons\ 0\ nil)\} \\ m_R & = & \lambda s{:}\{y : natlist\}. \\ & & \quad \{getcount = length(s.y) \\ & & \quad ,count = \{y = cons\ 0\ s.y\} \\ & & \quad ,reset = \{y = cons\ 2\ (cons\ 2\ nil)\}\} \end{array}$$

$RCounterClass$ can be understood as the smallest subset of $RCounter$ that contains $\mathbf{object}\ \langle init_R, m_R\rangle$ and is closed under method invocation.

$RCounterClass$ can be viewed as a behavioural subtype of $CounterClass$, because, despite their different implementations, the objects in $RCounterClass$ behave just like objects in $CounterClass$. By invoking only the methods $count$ and $getcount$, it is not possible to distinguish the objects in $RCounterClass$ from those in $CounterClass$, since for every object $o' : RCounterClass$ there is a $o : CounterClass$ that is indistinguishable from o'. More formally, we can say that $RCounterClass$ is a behavioural subtype of $CounterClass$ because there exists a relation $\sim \subseteq \{x : Nat\} \times \{y : natlist\}$ such that

$$s \sim s' \Rightarrow \left\{ \begin{array}{c} (m\ s).getcount =_{Nat} (m_R\ s').getcount \\ (m\ s).count \sim (m_R\ s').count \end{array} \right. \qquad \text{(i)}$$

and

$$\begin{array}{l} \forall o'{:}RCounterClass. \\ \exists o{:}CounterClass. \\ \text{the states of } o \text{ and } o' \text{ are related by } \sim. \end{array} \qquad \text{(ii)}$$

This relation $\sim$ is of course

$$s \sim s' \iff s.x = length(s'.y)\ .$$

Together conditions (i) and (ii) guarantee that for every $o' : RCounterClass$ there is a $o' : CounterClass$ such that o' is indistinguishable from o, if we are only allowed to invoke their $getcount$ and $count$ methods.

Intuitively, the conditions above guarantee that objects in $RCounterClass$ have the properties that all objects in $CounterClass$ have, and maybe more. For instance, note that $o{\leftarrow}getcount \geq 2$ for all $o : RCounterClass$.

Examples of classes that are not behavioural subtypes of $CounterClass$ are

$$\begin{array}{rcl} CounterClass1 & = & (\mathbf{Class}\, CounterI \,\mathbf{with}\, init, m') \\ RCounterClass1 & = & (\mathbf{Class}\, CounterI \,\mathbf{with}\, init_R, m'_R) \end{array}$$

with

$$
\begin{aligned}
m' &= \lambda s{:}\{x : Nat\}.\, \{get = s.x, count = \{x = s.x + 2\}\} \\
m'_R &= \lambda s{:}\{y : natlist\}. \\
&\qquad \{getcount = length(s.y) \\
&\qquad , count = \{y = cons\ 0\ s.y\} \\
&\qquad , reset = \{y = nil\}\}
\end{aligned}
$$

These classes are not behavioural subtypes of $CounterClass$, for slightly different reasons. For $CounterClass1$ we cannot find a suitable simulation relation. For $RCounterClass1$ there is an obvious simulation relation, namely the same one we used for $RCounterClass$. However, there is a object in $RCounterClass1$ for which there is no related object in $CounterClass$: resetting a counter in $RCounterClass1$ produces an object $o \equiv \mathbf{object}\ \langle\{y = nil\}, m\rangle$ for which $o{\leftarrow}getcount = 0$, and this object cannot be related to any object in $CounterClass$, since all counters in $CounterClass$ have a $getcount$ greater then 0.

For class types, the interpretation of subtyping in the PER model – subtypes are subpers – turns out to be equivalent to the informal notion of behavioural subtyping introduced here, i.e. given by properties such as (i) and (ii). So in the PER model

$$
\begin{aligned}
[\![RCounterClass]\!] &\subseteq [\![Counterclass]\!] \\
[\![CounterClass1]\!] &\not\subseteq [\![Counterclass]\!] \\
[\![RCounterClass1]\!] &\not\subseteq [\![Counterclass]\!]
\end{aligned}
$$

The fact that the PER model already provides bisimulation as the notion of equality plays a vital role here.

3 Definition of λ^{OO}

The raw syntax of the *terms* a, *types* A and *signatures* I of λ^{OO} is given by the following grammar

$$
\begin{aligned}
a &::= x \mid \lambda x{:}A.\, a \mid aa \mid \{l_1 = a, \ldots, l_n = a\} \mid a.l \mid \\
&\quad\;\; a{\leftarrow}l(a) \mid \mathbf{object}_I\ \langle A, a, a\rangle \\
A &::= X \mid A{\rightarrow}A' \mid \{l_1{:}A, \ldots, l_n{:}A\} \mid \\
&\quad\;\; \mathbf{Sig}\ I \mid \mathbf{Class}\ I\ \mathbf{with}\ A, a, a \\
I &::= (X)\{l_1{:}A{\rightarrow}A, \ldots, l_n{:}A{\rightarrow}A\}
\end{aligned}
$$

Here x ranges over *term variables*, X ranges over *type variables*, and l over a countable set $\mathcal{L}$ of *labels*. The type variable X is bound in $(X)A$, and if $I \equiv (X)A$ we write $I(B)$ for $[B/X]A$, where $[B/X]A$ denotes the capture-free substitution of B for X in A. Expressions equal up to the names of bound variables and permutation of fields are identified as usual, and we assume that the same label never occurs twice in a record (type) or interface.

For simplicity, we divide $\mathcal{L}$ into a set $\mathcal{L}_{obs}$ of labels that can be used as names for *observer methods* and a set $\mathcal{L}_{mut}$ of labels that can be used as names for *mutator methods*, and we insist that

$$
\begin{aligned}
l_i \in \mathcal{L}_{obs} &\Rightarrow X \notin \mathsf{FV}(A_i{\rightarrow}B_i) \\
l_i \in \mathcal{L}_{mut} &\Rightarrow X \notin \mathsf{FV}(A_i) \wedge X \equiv B_i
\end{aligned}
$$

for any signature $(X)\{l_1{:}A_1{\rightarrow}B_1, , \ldots, l_n{:}A_n{\rightarrow}B_n\}$.

The *contexts* of λ^{OO} are given by

$$
\Gamma \quad ::= \quad \epsilon \mid \Gamma, x : A
$$

with no variable occurring twice.

The subtyping and typing rules of λ^{OO} are given in tables 1 and 3. The rules for well-formedness of contexts and types are given in Table 2. These are needed because there are types with terms as subexpressions, namely the class types. Only the well-formedness rule for class types is given, for the other are trivial. (E.g., $A{\rightarrow}B$ is well-formed in Γ if A and B are well-formed in Γ, etc.)

The *reduction relation* $\rhd$ on terms is given by the rules

$$
\begin{aligned}
(\lambda x{:}A.\, b)a &\rhd [a/x]b \\
\{l_1 = a_1, \ldots, l_n = a_n\}.l_i &\rhd a_i \\
(\mathbf{object}_I\ \langle s, m\rangle){\leftarrow}l(a) &\rhd (m\ s).l\ a \\
&\quad\;\, \text{if } l \in \mathcal{L}_{obs} \\
(\mathbf{object}_I\ \langle s, m\rangle){\leftarrow}l(a) &\rhd \mathbf{object}_I\ \langle (m\ s).l\ a, m\rangle \\
&\quad\;\, \text{if } l \in \mathcal{L}_{mut}
\end{aligned}
$$

where $[a/x]b$ denotes the capture-free substitution of a for x in b.

We write $\simeq$ for the reflexive, transitive, and symmetric closure of $\rhd$, and $\Gamma \vdash a \simeq a' : A$ for $\Gamma \vdash a : A \wedge \Gamma \vdash a' : A \wedge a \simeq a'$.

4 PER Model for λ^{OO} without Class Types

As we mentioned earlier, with the exception of the class types, λ^{OO} just introduces some syntactical sugar for the existential object encoding of objects in $F^{\omega}_{\leq}$ given in [PT94]. In fact, we only need the subsystem $F_{\leq}$ of $F^{\omega}_{\leq}$ as target system.[1] So any model of $F_{\leq}$ can be used as model for λ^{OO} without classes.

The model we use is the PER model of [PAC94]. This model combines the PER models of [BL90] and [BFAS90]: it is essentially just the model of [BL90] – and interprets subtypes as subpers –, but it uses the interpretation of polymorphic types given in [BFAS90] – rather than the more standard one used in [BL90] – to ensure parametricity. The vital property of this model that we are interested in is that the existential types $\exists X.\, X \times (X{\rightarrow}I(X))$ are interpreted as *final coalgebras*, as is proved in [Has94] and [PA93].

First we make precise how λ^{OO} without class types can be regarded as syntactic sugar for $F_{\leq}$.

As far as types are concerned the types, interface types abbreviate existential types:

$$
\mathbf{Sig}\, I \quad = \quad \exists X.\, X \times (X{\rightarrow}I(X)).
$$

Remark 4.1 Here $\times$ and $\exists$ stand for the usual $F_{\leq}$ encodings. The record types we have in λ^{OO} are not part of $F_{\leq}$, but these can be encoded in $F_{\leq}$ using the trick of [Car92]: We assume there is some enumeration of the labels and we represent the record type $\{l_1 : A_1, \ldots, l_n : A_n\}$ by the product $B_1 \times \ldots B_m$ where m is the greatest index of the l_i and $B_j = A_i$ if j is the index of l_i and $B = Top$ if j is not the index of any of the l_i. Product types can of course be encoded in $F_{\leq}$ in the usual way.

The encodings of existential types, product types and record types will be left implicit, so we use the normal notation for existentials, products, and records as abbreviation for their $F_{\leq}$-encodings.

[1] In [PT94] $F^{\omega}_{\leq}$ rather than $F_{\leq}$ is needed to type generic method invocations, i.e. method invocations not yet applied to a particular object, which in our syntax would be written as "$\leftarrow l$". We don't have these in λ^{OO}.

$$\frac{}{A \leq A}\ \leq\text{-refl} \qquad \frac{A \leq A' \quad A' \leq A''}{A \leq A''}\ \leq\text{-trans} \qquad \frac{A_1 \leq B_1 \ldots A_n \leq B_n \quad m \geq n}{\{l_1{:}A_1, \ldots, l_n{:}A_m\} \leq \{l_1{:}B_1, \ldots, l_n{:}B_n\}}\ \leq\text{-record}$$

$$\frac{A' \leq A \quad B \leq B'}{A \rightarrow B \leq A' \rightarrow B'}\ \leq\text{-}\rightarrow \qquad \frac{A \leq B}{\textbf{Sig}\ (X)A \leq \textbf{Sig}\ (X)B}\ \leq\text{-sig} \qquad \frac{}{(\textbf{Class}\ I\ \textbf{with}\ meth, init) \leq \textbf{Sig}\ I}\ \leq\text{-class}$$

Table 1. Subtyping

$$\frac{}{\epsilon \vdash ok}\ \text{empty-ok} \qquad \frac{\Gamma \vdash ok \quad \Gamma \vdash A\ ok}{\Gamma, x : A \vdash ok}\ \text{weaken-ok} \qquad \frac{\Gamma ok \quad \Gamma \vdash init : Rep \quad \Gamma \vdash m : Rep \rightarrow I(Rep)}{\Gamma \vdash (\textbf{Class}\ I\ \textbf{with}\ s, m)\ ok}\ \text{class-ok}$$

Table 2. Well-formedness

$$\frac{\Gamma, x : A, \Gamma' \vdash ok}{\Gamma, x : A, \Gamma' \vdash x : A}\ \text{var} \qquad \frac{\Gamma \vdash a : A \quad A \leq B}{\Gamma \vdash a : B}\ \text{sub} \qquad \frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda x{:}A.\, b : A \rightarrow B}\ \rightarrow\text{-I} \qquad \frac{\Gamma \vdash f : A \rightarrow B \quad \Gamma \vdash a : A}{\Gamma \vdash fa : B}\ \rightarrow\text{-E}$$

$$\frac{\Gamma \vdash a_1 : A_1 \quad \ldots \quad \Gamma \vdash a_n : A_n}{\Gamma \vdash \{l_1 = a_1, \ldots, l_n = a_n\} : \{l_1{:}A_1, \ldots, l_n{:}A_n\}}\ \text{record-I} \qquad \frac{\Gamma \vdash a : \{l_1{:}A_1, \ldots, l_n{:}A_n\} \quad l_i \in \{l_1, \ldots, l_n\}}{\Gamma \vdash a.l_i : A_i}\ \text{record-E}$$

$$\frac{\Gamma \vdash s : Rep \quad \Gamma \vdash m : Rep \rightarrow I(Rep)}{\Gamma \vdash \textbf{object}_I\ \langle s, m\rangle : \textbf{Sig}\ I}\ \text{object-I1} \qquad \frac{\Gamma \vdash s : Rep \quad \Gamma \vdash m : Rep \rightarrow I(Rep)}{\Gamma \vdash \textbf{object}_I\ \langle s, m\rangle : (\textbf{Class}\ I\ \textbf{with}\ s, m)}\ \text{object-I2}$$

For $SC \equiv \textbf{Sig}\ I$ or $SC \equiv (\textbf{Class}\ I\ \textbf{with}\ meth, init)$ with $I(X) = \{\ldots, l : A \rightarrow B, \ldots\}$:

$$\frac{\Gamma \vdash o : SC \quad \Gamma \vdash a : A \quad l \in \mathcal{L}_{obs}}{\Gamma \vdash o \leftarrow l(a) : B}\ \text{object-E-obs} \qquad \frac{\Gamma \vdash o : SC \quad \Gamma \vdash a : A \quad l \in \mathcal{L}_{mut}}{\Gamma \vdash o \leftarrow l(a) : SC}\ \text{object-E-mut}$$

Table 3. Typing

The λ^{OO} syntax for object formation and method invocation can be interpreted by the following $F_{\leq}$-terms:

$$\begin{array}{rcl} \textbf{object}_I\ \langle s, m\rangle & = & \textbf{pack}\ \langle Rep, (s, m)\rangle\ \textbf{to}\ \textbf{Sig}\ I \\ o \leftarrow l(a) & = & \textbf{open}\ o\ \textbf{as}\ \langle X, (s, m)\rangle\ \textbf{in}\ (m\ s).l\ a \\ & & \quad \text{if } l \in \mathcal{L}_{obs} \\ o \leftarrow l(a) & = & \textbf{open}\ o\ \textbf{as}\ \langle X, (s, m)\rangle\ \textbf{in} \\ & & \quad (\textbf{pack}\ \langle X, (m\ s).l\ a, m)\rangle\ \textbf{to}\ \textbf{Sig}\ I) \\ & & \quad \text{if } l \in \mathcal{L}_{mut} \end{array}$$

Note that some type information is missing in the λ^{OO}-terms, namely the type Rep of s in the first clause and $\textbf{Sig}\ I$ in the last clause. However, type information in terms does not play a role in the interpretation of terms in the PER model that we are interested in, so we can safely ignore this.

We now consider the PER model of [PAC94]. Here types are interpreted as partial equivalence relations (pers) on $\mathbb{N}$, and terms are interpreted as natural numbers, using some enumeration of the partial recursive functions.

The per interpreting a type A is written as $[\![A]\!]_\xi$, where ξ is a type environment that maps from type variables to pers. The natural number interpreting a term a is written as $[\![a]\!]_\eta$, where η is a term environment that maps terms to natural numbers. Another way of looking at the PER model is to interpret a type A as the set of $[\![A]\!]_\xi$-equivalence classes, and to interpret a term $a : A$ as the $[\![A]\!]_\xi$-equivalence class that contains $[\![a]\!]_\eta$.

λ^{OO} without class types can be interpreted in this PER model for $F_{\leq}$ by interpreting interface types, objects, and method invocations by their $F_{\leq}$-counterparts:

$$\begin{array}{rcl} [\![\textbf{Sig}\ I]\!]_\xi & = & [\![\exists X.\, X \times (X \rightarrow I(X))]\!]_\xi \\ [\![\textbf{object}_I\ \langle s, m\rangle]\!]_\eta & = & [\![\textbf{pack}\ \langle Rep, (s, m)\rangle\ \textbf{to}\ \textbf{Sig}\ I]\!]_\eta \\ [\![o \leftarrow l(a)]\!]_\eta & = & [\![\textbf{open}\ o\ \textbf{as}\ \langle X, (s, m)\rangle\ \textbf{in}\ (m\ s).l\ a]\!]_\eta \\ & & \quad \text{if } l \in \mathcal{L}_{obs} \\ [\![o \leftarrow l(a)]\!]_\eta & = & [\![\textbf{open}\ o\ \textbf{as}\ \langle X, (s, m)\rangle\ \textbf{in} \\ & & \quad \textbf{pack}\ \langle X, (m\ s).l\ a, m)\rangle\ \textbf{to}\ \textbf{Sig}\ I]\!]_\eta \\ & & \quad \text{if } l \in \mathcal{L}_{mut} \end{array}$$

A pair (η, ξ) satisfies a context Γ – written $(\eta, \xi) \models \Gamma$ – iff $(\eta(x), \eta(x)) \in [\![A]\!]_\xi$ for all declarations $x : A$ in Γ. Given that the PER model is a sound model for $F_{\leq}$, it is also a sound model for λ^{OO}:

Theorem 4.2
If $\Gamma \vdash a \simeq a' : A$ in λ^{OO} without class types, then $([\![a]\!]_\eta, [\![a']\!]_\eta) \in [\![A]\!]_{\eta,\xi}$ for all $(\eta, \xi) \models \Gamma$.

We will now look at the interpretations of objects and interface types in more detail. Some properties of these interpretations will be used to interpret class types in the next section.

First a few definitions. We write $n \cdot m$ for the n^{th} partial recursive function applied to m, and $\lambda\!\!\lambda x.\, E(x)$ for the index of a partial recursive function for which $\lambda\!\!\lambda x.\, E(x) \cdot n = E(n)$, where $E(x)$ is a partial recursive description of a natural number depending on x.

We write $[n]_R$ for the R-equivalence containing n, and $\mathbb{N}/R$ for the set of R-equivalence classes. For pers R and S, the per $R \twoheadrightarrow S$ is defined as

$$R \twoheadrightarrow S = \{(f, f') \in \mathbb{N} \times \mathbb{N} \mid \forall (a, a') \in A.\, (f \cdot a, f' \cdot a') \in B\}.$$

We write $n \mathrel{\underline{\in}} R$ as abbreviation for $(n,n) \in R$.

The category PER is defined as in [BFAS90] and [Has94] as the category with as objects pers on $\mathbb{N}$ and as the arrows from R to S total functions from $\mathbb{N}/R$ to $\mathbb{N}/S$ named by a partial recursive functions in $R \twoheadrightarrow S$.

4.1 Interpretation of objects

We define $\mathsf{obj}\langle\_,\_\rangle$ and $\_ \Leftarrow_l \_$ as the interpretations of **object** $\langle\_,\_\rangle$ and $\_ \leftarrow l(\_)$ in the PER model. So

$$[\![\mathbf{object}_I\ \langle s,m\rangle]\!]_\eta \quad = \quad \mathsf{obj}\langle [\![s]\!]_\eta, [\![m]\!]_\eta \rangle$$

and

$$[\![o \leftarrow l(a)]\!]_\eta \quad = \quad [\![o]\!]_\eta \Leftarrow_l [\![a]\!]_\eta \ .$$

Looking at the interpretations of the $F_{\leq}$-terms that **object** $\langle\rangle$ and $\leftarrow l$ denote, we find that

$$(\mathsf{obj}\langle s,m\rangle) \Leftarrow_l a \quad = \quad \begin{cases} (m \cdot s).l \cdot a & \text{if } l \in \mathcal{L}_{obs} \\ \mathsf{obj}\langle (m\cdot s).l\cdot a, m\rangle & \text{if } l \in \mathcal{L}_{mut} \end{cases}$$

Note that we abuse our notation for field selection here and write ".l" for the selection of the l-field, when we should really use the interpretation of the l-projection under the encoding of records as products discussed in Remark 4.1.

4.2 Interpretation of interface types

The interesting property of the PER model is that the existential types of the form $\exists X.\, X \times (X \to I(X))$ – i.e. interface types – are interpreted as final coalgebras, as is proved in [Has94] and [PA93].

We define $\mathsf{SIG}(\_)$ as the interpretation of **Sig** $(\_)$ in the PER model. So

$$\begin{aligned} [\![\mathbf{Sig}\ I]\!]_\xi &= [\![\exists X.\, X \times (X \to I(X))]\!]_\xi \\ &= \mathsf{SIG}([\![I]\!]_\xi) \end{aligned}$$

Let $\mathcal{I} : \mathsf{PER} \to \mathsf{PER}$ be the functor interpreting some interface I in some environment ξ, i.e. $\mathcal{I} = [\![I]\!]_\xi$. We define $\mathsf{out}_{\mathcal{I}}$ as

$$\mathsf{out}_{\mathcal{I}} \quad = \quad \lambda\!\!\lambda o.\, \mathcal{I}(\lambda\!\!\lambda s.\, \mathsf{obj}\langle s, \mathsf{snd}(o)\rangle) \cdot (\mathsf{snd}(o) \cdot \mathsf{fst}(o)) \ .$$

Another way of defining $\mathsf{out}_{\mathcal{I}}$ is as the interpretation of $out_I : \mathbf{Sig}\ I \to I(\mathbf{Sig}\ I)$, defined as follows

$$\begin{aligned} out_I \quad &= \quad \lambda o{:}\mathbf{Sig}\ I.\ \mathbf{open}\ o\ \mathbf{as}\ \langle X, (s,m)\rangle\ \mathbf{in} \\ &\qquad\quad I_m(\lambda x{:}X.\, \mathbf{object}_I\ \langle x, m\rangle)\ (m\ s) \\ &: \quad \mathbf{Sig}\ I \to I(\mathbf{Sig}\ I) \end{aligned}$$

Here I_m is the action of I on functions, with $I_m(f) : I(A) \to I(B)$ for any $f : A \to B$, defined in the usual way by induction on I.

The pair $(\mathsf{SIG}(\mathcal{I}), \mathsf{out}_{\mathcal{I}})$ is a final coalgebra [Has94][PA93]. So

$$\mathsf{out}_{\mathcal{I}} \mathrel{\underline{\in}} \mathsf{SIG}(\mathcal{I}) \twoheadrightarrow \mathcal{I}(\mathsf{SIG}(\mathcal{I})),$$

and we have the following properties;

Property 4.3 *Let $s_i \mathrel{\underline{\in}} Rep_i$ and $m_i \mathrel{\underline{\in}} Rep_i \twoheadrightarrow \mathcal{I}(Rep_i)$ for certain pers Rep_i, $i = 1,2$.*

Then

$$\begin{aligned} & (\mathsf{obj}\langle s_1, m_1\rangle, \mathsf{obj}\langle s_2, m_2\rangle) \in \mathsf{SIG}(\mathcal{I}) \\ \iff \quad & \exists \sim \in \mathbb{N} \times \mathbb{N}.\ \ \sim = Rep_1; \sim; Rep_2\ \wedge \\ & \qquad s_1 \sim s_2\ \wedge \\ & \qquad (m_1, m_2) \in \sim \twoheadrightarrow \mathcal{I}(\sim) \end{aligned}$$

Here ; denotes composition of relations.

Property 4.4 *For any relation $\sim \subseteq \mathbb{N} \times \mathbb{N}$*

$$\mathsf{out}_{\mathcal{I}} \mathrel{\underline{\in}} \sim \twoheadrightarrow \mathcal{I}(\sim) \ \Rightarrow\ \sim \subseteq \mathsf{SIG}(\mathcal{I}).$$

In other words, $\mathsf{SIG}(\mathcal{I})$ is the maximum bisimulation.

These properties are particular cases of Theorems 7 and 11 in [PA93]. A direct consequence (take $\sim = Rep_i = Rep$) of Property 4.3 is:

Corollary 4.5 *Let $(s_1, s_2) \in Rep$ and $(m_1, m_2) \in Rep \twoheadrightarrow I(Rep)$ for some per Rep.*

Then $(\mathsf{obj}\langle s_1, m_1\rangle, \mathsf{obj}\langle s_2, m_2\rangle) \in \mathsf{SIG}(\mathcal{I})$.

The mapping $\mathsf{out}_{\mathcal{I}}$ is related to the interpretation of method invocations $\_ \Leftarrow_l \_$ as follows:

Property 4.6 *Let $\mathcal{I} = [\![I]\!]_\xi$ for some ξ, with I an interface that contains a method l. Then*

$$o \Leftarrow_l a \quad = \quad (\mathsf{out}_{\mathcal{I}} \cdot o).l \cdot a$$

As before, we abuse our notation for field selection here.

5 Model for λ^{OO} with Class Types

We now extend the interpretation of λ^{OO} without class types in the PER model of [PAC94] to an interpretation of the full λ^{OO} including the class types.

Definition 5.1 *The relation $\sqsubseteq$ on pers is defined by*

$$R \sqsubseteq S \quad \iff \quad \mathbb{N}/R \subseteq \mathbb{N}/S.$$

An equivalent definition is

$$R \sqsubseteq S \quad \iff \quad R \subseteq S \wedge R = S;R;S\ .$$

The relation $\sqsubseteq$ is used to define the interpretation of a class types:

Definition 5.2 *For $\mathcal{I} : \mathsf{PER} \to \mathsf{PER}$ and $m, n \in \mathbb{N}$ we define $\mathsf{CLASS}(\mathcal{I}, s, m)$ as follows*

$$\begin{aligned} & \mathsf{CLASS}(\mathcal{I}, s, m) \\ = \quad & \\ & \bigcap\{X \sqsubseteq \mathsf{SIG}(\mathcal{I}) \mid \mathsf{obj}\langle s,m\rangle \mathrel{\underline{\in}} X \wedge \mathsf{out}_{\mathcal{I}} \mathrel{\underline{\in}} X \twoheadrightarrow \mathcal{I}(X)\}. \end{aligned}$$

This defines $\mathbb{N}/\mathsf{CLASS}(\mathcal{I}, s, m)$ as the smallest subset of $\mathbb{N}/\mathsf{SIG}(\mathcal{I})$ that contains $[\mathsf{obj}\langle s,m\rangle]_{\mathsf{SIG}(\mathcal{I})}$ and is closed under method invocations.

Lemma 5.3 *Let $\mathcal{I} : \mathsf{PER} \to \mathsf{PER}$. Suppose that $\mathcal{I}$ is continuous – i.e. $\mathcal{I}(\bigcap_i X_i) = \bigcap_i \mathcal{I}(X_i)$ – and suppose that $\mathcal{I}(R); \mathcal{I}(S) \subseteq \mathcal{I}(R;S)$ for all pers R and S. Then*

1. $\mathsf{CLASS}(\mathcal{I}, s, m) \sqsubseteq \mathsf{SIG}(\mathcal{I})$

2. $\mathsf{obj}\langle s,m\rangle \sqsubseteq \mathsf{CLASS}(\mathcal{I},s,m)$

3. $out_I \sqsubseteq \mathsf{CLASS}(\mathcal{I},s,m) \twoheadrightarrow I(\mathsf{CLASS}(\mathcal{I},s,m))$

4. *Let* $(s_1,s_2) \in Rep$ *and* $(m_1,m_2) \in Rep \twoheadrightarrow \mathcal{I}(Rep)$. *Then* $\mathsf{CLASS}(\mathcal{I},s_1,m_1) = \mathsf{CLASS}(\mathcal{I},s_2,m_2)$.

Proof. These properties easily follow from the definition of CLASS and the assumptions on $\mathcal{I}$. For *4.* use Corollary 4.5 to deduce that

$$(s,m) \sqsubseteq X \Longleftrightarrow (s',m') \sqsubseteq X \qquad \text{for any } X \sqsubseteq \mathsf{SIG}(I)$$

from the assumptions on s_i and m_i. □

It is easy to verify that any $\mathcal{I}$ that is the interpretation of a λ^{OO}-signature will satisfy the conditions of the lemma above.

CLASS will now be used to extend the PER model of λ^{OO} without class types to a model for the full λ^{OO}. The definition of this model is given below.

As far as terms is concerned nothing changes. In λ^{OO} without class types we have the same terms as in λ^{OO} with class type, so the terms can be interpreted as in the PER model discussed in the previous section:

Definition 5.4 *The interpretation* $[\![a]\!]_\eta \in \mathbb{N}$ *of a* λ^{OO}*-term* M *in term environment* η *is defined as*

$$[\![a]\!]_\eta = [\![Erase(a)]\!]_\eta\,,$$

where

$$\begin{array}{rcl}
Erase(\mathbf{object}_I\ \langle s,m\rangle) &=& \mathbf{object}\ \langle Erase(s), Erase(m)\rangle \\
Erase(o \leftarrow l(a)) &=& Erase(o) \leftarrow l(Erase(a)) \\
Erase(\lambda x{:}A.\, b) &=& \lambda x.\ Erase(b) \\
Erase(fa) &=& Erase(f)Erase(a) \\
Erase(a.l) &=& Erase(a).l \\
\multicolumn{3}{l}{Erase(\{l_1 = a_1,\ldots,l_n = a_n\}) =} \\
&& \{l_1 = Erase(a_1),\ldots,l_n = Erase(a_n)\}
\end{array}$$

and he interpretation of erased terms is defined by

$$\begin{array}{rcl}
[\![x]\!]_\eta &=& \eta(x) \\
[\![\lambda x.\, b]\!]_\eta &=& \lambda\!\!\lambda n.\ [\![b]\!]_{[n/x]\eta} \\
[\![fa]\!]_\eta &=& [\![f]\!]_\eta \cdot [\![a]\!]_\eta \\
[\![\mathbf{object}_I\ \langle s,m\rangle]\!]_\eta &=& \mathsf{obj}\langle [\![s]\!]_\eta, [\![m]\!]_\eta\rangle \\
[\![o \leftarrow l(a)]\!]_\eta &=& [\![o]\!]_\eta \Leftarrow_l [\![a]\!]_\eta \\
[\![\{l_1 = a_1,\ldots,l_n = a_n\}]\!]_\eta &=& \{l_1 = [\![a_1]\!]_\eta,\ldots,l_n = [\![a_n]\!]_\eta\} \\
[\![a.l]\!]_\eta &=& [\![a]\!]_\eta\, .l
\end{array}$$

Here we again abuse our notation for records and field selection as shorthand for their interpretations under the encoding discussed in Remark 4.1.

Because class types contain terms as subexpressions, the interpretation of types now has to be given w.r.t. a term environment η as well as a type environment ξ :

Definition 5.5 *The interpretation* $[\![A]\!]_{\eta,\xi}$ *of a type* A *in environment* (η,ξ) *is given by*

$$\begin{array}{rcl}
[\![X]\!]_{\eta,\xi} &=& \xi(X) \\
[\![A \rightarrow B]\!]_{\eta,\xi} &=& [\![A]\!]_{\eta,\xi} \twoheadrightarrow [\![B]\!]_{\eta,\xi} \\
[\![\{l_1{:}A_1,\ldots,l_n{:}A_n\}]\!]_{\eta,\xi} &=& \{l_1 = [\![A_1]\!]_{\eta,\xi},\ldots,l_n = [\![A_n]\!]_{\eta,\xi}\} \\
[\![\mathbf{Sig}\ I]\!]_{\eta,\xi} &=& \mathsf{SIG}([\![I]\!]_{\eta,\xi}) \\
[\![\mathbf{Class}\ I\ \mathbf{with}\ init, m]\!]_{\eta,\xi} &=& \mathsf{CLASS}([\![I]\!]_{\eta,\xi}, [\![init]\!]_\eta, [\![m]\!]_\eta)
\end{array}$$

Again, the notation for record types is abused as shorthand for their interpretations under the encoding discussed in Remark 4.1.

Theorem 5.6
If $\Gamma \vdash a \simeq a' : A$ *in* λ^{OO}
then $([\![a]\!]_\eta, [\![a']\!]_\eta) \in [\![A]\!]_{\eta,\xi}$ *for all* $(\eta,\xi) \models \Gamma$.

Proof. Soundness of type assignment, i.e.

$$\Gamma \vdash a : A \ \wedge\ (\eta,\xi) \models \Gamma \Rightarrow [\![a]\!]_\eta \sqsubseteq [\![A]\!]_{\eta,\xi},$$

can be proved in the usual way. Lemma 5.3.1 is needed for soundness of the subtyping rule for classes, 5.3.2 for soundness of the introduction rule for classes, and 5.3.3 – together with Lemma 4.6 – for soundness of the elimination rules for classes.

No extra work is needed to prove soundness of reduction: we can reuse the following property of the PER-interpretation of $F_\leq$: if $[\![a]\!]_\eta$ and $[\![a']\!]_\eta$ are defined, then

$$a =_\beta a' \Rightarrow [\![a]\!]_\eta = [\![a']\!]_\eta.$$

Since the mapping from λ^{OO} to $F_\leq$ preserves reduction, we immediately have the property that if $[\![a]\!]_\eta$ and $[\![a']\!]_\eta$ are defined, then

$$a \simeq a' \Rightarrow [\![a]\!]_\eta = [\![a']\!]_\eta.$$ □

5.1 Subtyping is behavioural subtyping

We now show that in the PER model subtyping between class types corresponds with the notion of behavioral subtyping as we informally explained it in Section 2.

Definition 5.7 *For* $init \sqsubseteq Rep$ *and* $m \sqsubseteq Rep \twoheadrightarrow (Rep)$ *the per* $\mathsf{REACH}(\mathcal{I}, Rep, init, m)$ *is defined as follows:*

$$\begin{array}{l}
\mathsf{REACH}(\mathcal{I}, Rep, init, m) \\
= \\
\bigcap\{X \sqsubseteq Rep \mid init \sqsubseteq X \ \wedge\ m \sqsubseteq X \twoheadrightarrow \mathcal{I}(X)\}.
\end{array}$$

$\mathbb{N}/\mathsf{REACH}(\mathcal{I}, Rep, init, m)$ is the set of those Rep-equivalence classes reachable from the state *init* using the method implementations m. Note the similarity between the definition of REACH and the definition of CLASS. There is close relationship between the two:

Lemma 5.8 *Let* $\mathcal{I} : \mathsf{PER} \rightarrow \mathsf{PER}$. *Suppose that* $\mathcal{I}$ *is continuous – i.e.* $\mathcal{I}(\bigcap_i X_i) = \bigcap_i \mathcal{I}(X_i)$ *– and that* $\mathcal{I}(R);\mathcal{I}(S) \subseteq \mathcal{I}(R;S)$ *for all pers* R *and* S. *Then*

$$\begin{array}{l}
\mathbb{N}/\mathsf{CLASS}(\mathcal{I}, init, m) \\
= \\
\{[\mathsf{obj}\langle s,m\rangle]_{\mathsf{SIG}(\mathcal{I})} \mid s \sqsubseteq \mathsf{REACH}(\mathcal{I}, Rep, init, m)\}
\end{array}$$

Proof. (Sketch) First we consider ($\subseteq$). Define the per X as

$$X = Rep\ \cap\ (f; \mathsf{CLASS}(\mathcal{I}, init, m); f^{\leftarrow}),$$

where $f \subseteq \mathbb{N} \times \mathbb{N}$ is the relation $\{(s, \mathsf{obj}\langle s,m\rangle) \mid s \in \mathbb{N}\}$ and $f^{\leftarrow}$ its inverse. We can prove the following properties of X:

- $X \sqsubseteq Rep$,
- $init \sqsubseteq X$,

- $m \sqsubseteq X \twoheadrightarrow \mathcal{I}(X)$.

It then follows by the definition of REACH that

$$\mathsf{REACH}(\mathcal{I}, Rep, init, m) \subseteq X \ ,$$

and so

$$\begin{array}{ll} & f^{\leftarrow}; \mathsf{REACH}(\mathcal{I}, Rep, init, m); f \\ \subseteq & f^{\leftarrow}; X; f \\ = & f^{\leftarrow};\ (Rep \ \cap \ f; \mathsf{CLASS}(\mathcal{I}, init, m); f^{\leftarrow})\ ; f \\ \subseteq & (f^{\leftarrow}; Rep; f) \ \cap \ (f^{\leftarrow}; f; \mathsf{CLASS}(\mathcal{I}, init, m); f^{\leftarrow}; f) \\ = & (f^{\leftarrow}; Rep; f) \ \cap \ \mathsf{CLASS}(\mathcal{I}, init, m) \\ \subseteq & \mathsf{CLASS}(\mathcal{I}, init, m) \end{array}$$

and ($\subseteq$) follows directly from the inclusion above.

Now to prove ($\supseteq$). Define

$$Y = \mathsf{SIG}(\mathcal{I}); f^{\leftarrow}; \mathsf{REACH}(\mathcal{I}, Rep, init, m); f; \mathsf{SIG}(\mathcal{I})$$

For Y we can prove the following properties:

- $Y \sqsubseteq \mathsf{SIG}(\mathcal{I})$,
- $(init, m) \sqsubseteq Y$,
- $out_{\mathcal{I}} \sqsubseteq Y \twoheadrightarrow \mathcal{I}(Y)$.

It then follows by the definition of CLASS that

$$\mathsf{CLASS}(\mathcal{I}, init, m) \subseteq Y \ ,$$

from which we can prove ($\supseteq$). □

The relation below defines subtyping between interpretations of signatures:

Definition 5.9 *The relation $\leq$ on PER→PER is defined as follows:*

$$\begin{array}{ll} & I_1 \leq I_2 \\ \iff & \\ & \mathcal{I}_1(X) \subseteq \mathcal{I}_2(X) \textit{ for all pers } X \textit{ and} \\ & \textit{both } \mathcal{I}_i \textit{ are continuous with} \\ & \mathcal{I}_i(R); \mathcal{I}_i(S) \textit{ for all pers } R \textit{ and } S. \end{array}$$

We can now state our main result, namely that, for interpretations of class types, the subset relation on pers is equivalent with the notion of behavioural subtyping that we described in Section 2.

Theorem 5.10 (Subtyping is Behavioural Subtyping)

Suppose $init_i \sqsubseteq Rep_i$ and $m_i \sqsubseteq Rep_i \twoheadrightarrow \mathcal{I}_i(Rep_i)$, for certain pers Rep_i for $i = 1, 2$. If $\mathcal{I}_1 \leq \mathcal{I}_2$ then

$$\begin{array}{l} \mathsf{CLASS}(\mathcal{I}_1, init_1, m_1) \subseteq \mathsf{CLASS}(\mathcal{I}_2, init_2, m_2) \\ \iff \\ \exists \sim \subseteq \mathbb{N} \times \mathbb{N}.\ \ \sim = Rep_1; \sim; Rep_2 \ \wedge \\ \qquad (m_1, m_2) \in \sim \twoheadrightarrow \mathcal{I}_2(\sim) \ \wedge \\ \qquad \forall s_1 \sqsubseteq \mathsf{REACH}(\mathcal{I}_1, Rep_1, init_1, m_1). \\ \qquad\ \ \exists s_2 \sqsubseteq \mathsf{REACH}(\mathcal{I}_2, Rep_2, init_2, m_2).\ s_1 \sim s_2 \end{array}$$

The second part of this theorem is a formal definition of the notion of behavioural subtyping discussed in Section 2: The condition

$$(m_1, m_2) \in \sim \twoheadrightarrow \mathcal{I}_2(\sim)$$

corresponds to condition (i) on page 3, and the condition

$$\begin{array}{l} \forall s_1 \sqsubseteq \mathsf{REACH}(\mathcal{I}_1, Rep_1, init_1, m_1). \\ \ \ \exists s_2 \sqsubseteq \mathsf{REACH}(\mathcal{I}_2, Rep_2, init_2, m_2).\ s_1 \sim s_2 \end{array}$$

corresponds to condition (ii) on page 3.

Proof. (Sketch) Define $C_i = \mathsf{CLASS}(\mathcal{I}_i, init_i, m_i)$ and $R_i = \mathsf{REACH}(\mathcal{I}_i, Rep_i, init_i, m_i)$.

($\Rightarrow$) Let $C_1 \subseteq C_2$.

Define $\sim \subseteq \mathbb{N} \times \mathbb{N}$ as follows

$$\sim = Rep_1; f_1; \mathsf{SIG}(\mathcal{I}); f_2^{\leftarrow}; Rep_2,$$

where $f_i = \{(s, \mathsf{obj}\langle s, m_i\rangle) \mid s \in \mathbb{N}\}$. For this relation $\sim$ the required properties can be proven.

($\Leftarrow$) Let $\sim \subseteq \mathbb{N} \times \mathbb{N}$ be such that

(i) $\sim = Rep_1; \sim; Rep_2$,

(ii) $(m_1, m_2) \in \sim \twoheadrightarrow \mathcal{I}_2(\sim)$,

(iii) $\forall s_1 \sqsubseteq R_1.\ \exists s_2 \sqsubseteq R_2.\ s_1 \sim s_2$.

Suppose that $o_1 \sqsubseteq C_1$. Then by Lemma 5.8($\subseteq$) there is an $s_1 \sqsubseteq R_1$ such that $(o, (s_1, m_1)) \in \mathsf{SIG}(\mathcal{I}_1)$. Then by (iii) there is an $s_2 \sqsubseteq R_2$ such that $s_1 \sim s_2$. By Property 4.3 it now follows that $(\mathsf{obj}\langle s_1, m_1\rangle, \mathsf{obj}\langle s_2, m_2\rangle) \in \mathsf{SIG}(\mathcal{I}_2)$, and then $(s_2, m_2) \sqsubseteq C_2$ by Lemma 5.8($\supseteq$). Moreover, by $\mathsf{SIG}(\mathcal{I}_1) \subseteq \mathsf{SIG}(\mathcal{I}_2)$ and the transitivity of pers: $(o_1, \mathsf{obj}\langle s_2, m_2\rangle) \in \mathsf{SIG}(\mathcal{I}_2)$.

This proves

$$o_1 \sqsubseteq C_1 \ \Rightarrow\ \exists o_2 \sqsubseteq C_2.\, (o_1, o_2) \in \mathsf{SIG}(\mathcal{I}_2).$$

From this property we can now deduce $C_1 \subseteq C_2$ using $C_i \sqsubseteq \mathsf{SIG}(\mathcal{I}_i)$ and some basic properties of $\sqsubseteq$. □

6 Conclusions and directions for future work

This paper establishes a link between three different strands of research on object-oriented languages, namely

- the type-theoretic approach to objects of [PT94],
- the work on behavioural subtyping of [Lea90],
- the categorical approach to objects of [Rei95].

For an extension of the type-theoretic encoding of object of Pierce and Turner [PT94] we have shown that the standard interpretation of subtyping in PER models – subtypes are subpers – provides exactly the notion of behavioural subtyping defined by Leavens [Lea90]. The crucial property is that object types are interpreted as final co-algebras. The correspondence between the existential object encoding and final coalgebras noted in [HP95] extends to our class types and sub-coalgebras of the final coalgebra. Sub-coalgebras are used in [Rei95] and [Jac96] as specifications of objects; our class types can of course be regarded as specifications, where we specify objects by giving a particular implementation.

The usefulness of the coalgebraic view of objects suggests that it might be better to use a primitive notion of coinductive type to present the existential object model, rather than an encoding of such types using existential types. The existential object model could for instance be carried out using Hagino's categorical datatypes [Hag87] extended with subtyping. (The interface types of λ^{OO} are essentially coalgebraic types in the sense of [Hag87].) An advantage would be that coinductive types only require a first-order type system, whereas existential types require a second-order type system.

One subject for future work is a more general description of a model for λ^{OO} in categorical terms, in which interface types are interpreted as final coalgebras, class types as subcoalgebras, and subtyping as coercions between them. We hope this will streamline much of the theory, and allow a presentation giving more than just sketches of proofs. (Note that I-coalgebras are only defined up to isomorphism, but the PER model here relies on the construction of a particular one of these as the interpretation of an interface type.)

We have not mentioned inheritance here. Inheritance for the existential model encoding is described in [PT94]. Now that we have a notion of behavioural subtyping, the interesting problem to look at is: "When does inheritance produce behavioural subtypes ?". Ideally we would want to formulate general conditions that are sufficient to guarantee that a class defined by inheritance is a behavioural subtype of the class it inherits from.

λ^{OO} could be extended with subtyping between class types, where this subtyping between class types is *declared* by the programmer. It would have to be the responsibility of the programmer that such declared subtyping is sound, as this is not something that can be decided by a typechecker. We would then really want a logic for reasoning about programs in which soundness of subtyping between class types can be expressed and (dis)proved. Such a logic would be an major topic for further investigation. Here it might be possible to use existing work on behavioural subtyping.

Finally, it would be interesting to see if the PER models of the other object encodings, e.g. those discussed in [BCP97], can also provide a notion of behavioural subtyping for class types. This would be more difficult: these other object encodings are in type systems with unrestricted recursion, and it is not clear what the effect of recursion would be. Also, the method updates allowed by some of these encodings would cause complications. These would have to be ruled out if we want to statically guarantee that all objects of a class type have the same method table.

References

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer, 1996.

[AL97] Martín Abadi and K. Rustan M. Leino. A logic of object-oriented programs. In *TAPSOFT '97: Theory and Practice of Software Development*, pages 682–696. Springer-Verlag, 1997.

[Ame89] Pierre America. A Behavioural Approach to Subtyping in Object-Oriented Languages. Technical Report Technical Report 443, Philips Research Laboratories, 1989.

[BCP97] Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. In *Theoretical Aspects of Computer Software (TACS'97), Sendai, Japan*, volume 1281, pages 415–438. Springer LNCS, September 1997.

[BFAS90] E.S. Bainbridge, P.J. Freyd, A.Scedrov, and P.J. Scott. Functorial polymorphism. *Theoretical Computer Science*, 70(1):35–64, 1990.

[BL90] Kim B. Bruce and Giuseppe Longo. A modest model of records, inheritance, and bounded quantification. *Information and Computation*, 87:196–240, 1990. Also in [GM94].

[Bru94] Kim B. Bruce. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 4(2), April 1994.

[Car88] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.

[Car92] Luca Cardelli. Extensible records in a pure calculus of subtyping. Research report 81, DEC Systems Research Center, 1992. Also in [GM94].

[FM94] Kathleen Fisher and John C. Mitchell. Notes on typed object-oriented programming. In *Proceedings of Theoretical Aspects of Computer Software (TACS'94) , Sendai, Japan*, volume 789 of *LNCS*, pages 844–886. Springer, 1994.

[FM97] K. Fisher and J.C. Mitchell. On the relationship between classes, objects, and data abstraction. In *Proceedings of the International Summer School on Mathematics of Program*, LNCS. Springer, 1997.

[GM94] Carl A. Gunter and John C. Mitchell. *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design.* The MIT Press, 1994.

[Hag87] Tatsuya Hagino. A typed lambda calculus with categorical type constructors. In D.H. Pitt, A Poigné, and D.E. Rydeheard, editors, *Category and Computer Science*, pages 140–157. Springer, September 1987.

[Has94] Tyu Hasegawa. Categorical data types in parametric polymorphism. *Mathematical Structures in Computer Science*, 4:71–109, 1994.

[HP95] Martin Hofmann and Benjamin C. Pierce. A unifying type-theoretic framework for objects. *Journal of Functional Programming*, 5(4):593–635, 1995.

[Jac96] Bart Jacobs. Objects and classes, coalgebraically. In *Object-Orientation with Parallelism and Persistence*, pages 83–103. Kluwer Academic Publishers, 1996. ISBN 0-7923-9770-3.

[Lea90] Gary T. Leavens. Modular verification of object-oriented programs with subtypes. Technical Report 90-09, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, July 1990.

[Lis88] Barbara H. Liskov. Data abstraction and hierarchy. *SIGPLAN Notices*, 23(3), 1988.

[LW94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *TOPLAS*, 16(6):1811–1841, November 1994.

[LW95] Gary T. Leavens and William E. Weihl. Specification and verification of object-oriented programs using supertype abstraction. *Acta Informatica*, 32(8):705–778, November 1995.

[Mau95] Ian Maung. On simulation, subtyping and substitutability in sequential object systems. *Formal Aspects of Computing*, 7(6):620–651, 1995.

[Mey88] Bertrand Meyer. *Object-oriented software construction*. Prentice Hall, 1988.

[MP88] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Trans. on Prog. Lang. and Syst.*, 10(3):470–502, 1988.

[PA93] Gordon Plotkin and Martin Abadi. A logic for parametric polymorphism. In *Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 361–375, 1993.

[PAC94] Gordon Plotkin, Martín Abadi, and Luca Cardelli. Subtyping and parametricity. In *Proceedings of the Ninth IEEE Symposium on Logic in Computer Science*, pages 310–319, 1994.

[PH97] A. Poetzsch-Heffter. *Specification and Verification of Object-Oriented Programs*. PhD thesis, Technische Universität München, 1997.

[PT94] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, April 1994.

[Rei95] Horst Reichel. An approach to object semantics based on terminal co-algebras. *Mathematical Structures in Computer Science*, 5:129–152, 1995.

Type Destructors

Martin Hofmann
Fachbereich Mathematik
Technische Hochschule Darmstadt
Schloßgartenstraße 7
D-64289 Darmstadt, Germany
mh@mathematik.th-darmstadt.de

Benjamin C. Pierce
Computer Science Department
Indiana University
Lindley Hall 215
Bloomington, IN 47405, USA
pierce@cs.indiana.edu

Abstract

We study a variant of System $F_{\leq}$ that integrates and generalizes several existing proposals for calculi with *structural typing rules.* To the usual type constructors ($\rightarrow$, $\times$, `All`, `Some`, `Rec`) we add a number of type *destructors*, each internalizing a useful fact about the subtyping relation. For example, in $F_{\leq}$ with products every closed subtype of a product $\mathtt{S}\times\mathtt{T}$ must itself be a product $\mathtt{S}'\times\mathtt{T}'$ with $\mathtt{S}'$<:$\mathtt{S}$ and $\mathtt{T}'$<:$\mathtt{T}$. We internalise this observation by introducing type destructors `.1` and `.2` and postulating an equivalence $\mathtt{T} =_\eta \mathtt{T.1}\times\mathtt{T.2}$ whenever `T` <: $\mathtt{U}\times\mathtt{V}$ (including, for example, when `T` is a variable). In other words, every subtype of a product type literally *is* a product type, modulo η-conversion.

Adding type destructors provides a clean solution to the problem of *polymorphic update* without introducing new term formers, new forms of polymorphism, or quantification over type operators. We illustrate this by giving elementary presentations of two well-known encodings of objects, one based on recursive record types and the other based on existential packages.

The formulation of type destructors poses some tricky meta-theoretic problems. We discuss two different variants: an "ideal" system where both constructors and destructors appear in general forms, and a more modest system, $F^D_{\leq}$, which imposes some restrictions in order to achieve a tractable metatheory.

1 Introduction

The search for type-theoretic foundations for object-oriented languages has driven the development of numerous typed lambda-calculi combining polymorphism and subtyping. The prototype of these systems is $F_{\leq}$ [CW85, CG92, CMMS94]. However, in the recent literature, many have observed that $F_{\leq}$ in its pure form is an inadequate framework for object-oriented programming. The problem is that the only way in which subtyping in $F_{\leq}$ can be used in typing terms is via the subsumption rule, which "wastes" some of the information contained in the subtyping relation. In particular, there is no way in $F_{\leq}$ to define *polymorphic update* functions—functions with types like `All(X<:T)X`$\rightarrow$`X` that do not behave like the polymorphic identity (or its approximations)—which play an important role in encodings of objects.

Presented at FOOL'5, 1998

To address this shortcoming, several extensions and refinements of $F_{\leq}$ have been proposed, including extensions to higher-order polymorphism [Car90, CL91, PT94, HP95b, PS94, Com94] and a number of special-purpose second-order systems, among them systems with record update [CM91, Car92, FM96, Pol96, HP95a], "structural unfolding" for recursive types [AC96], and "polymorphic repacking" for existential types [Pie96]. What the latter group of extensions have in common is that their soundness is intuitively argued for by using an internalisation of the generation lemma for subtyping: every concrete (closed) subtype of a concrete type `T` must have the same outermost type former as `T`.

A simple example in which this principle is applied (popularized by Cardelli [Car95], who attributes it to Abadi) is the following. If `X` <: $\mathtt{T}_1\times\mathtt{T}_2$, then, when `X` is eventually instantiated with a closed type, this type will be a product whose factors are subtypes of $\mathtt{T}_1$ and $\mathtt{T}_2$, respectively. So it should be sound to assume a function

$$\mathtt{mix} \quad \in \quad \mathtt{All(X{<:}Top{\times}Top)}\ \mathtt{X}\rightarrow\mathtt{X}\rightarrow\mathtt{X}$$

such that

$$\mathtt{mix}\ [\mathtt{S}_1\times\mathtt{S}_2]\ \mathtt{e}\ \mathtt{e}' \quad = \quad (\mathtt{e.1},\mathtt{e}'\mathtt{.2}).$$

That is, `mix` takes the first component of its first argument and the second component of its second argument to form a new element of type `X`. Clearly, this assumes that under the constraint `X` <: `Top`$\times$`Top` the variable `X` gets instantiated by a product and not by a base type or function type. Although the type system of $F_{\leq}$ ensures that this is indeed the case, $F_{\leq}$ provides no way to make use of this fact to define a function with `mix`'s behavior.

A more interesting example is a refinement of the standard unfolding rule for recursive types. It was used by Abadi, Cardelli, and Viswanathan [ACV96, AC96] to perform method calls in their encoding of objects:

$$\mathtt{unfold} \quad \in \quad \mathtt{All(X{<:}Rec(Z)T)}\ \mathtt{X}\rightarrow[\mathtt{X}/\mathtt{Z}]\mathtt{T}.$$

Here the idea is that (assuming monotone subtyping for recursive types) the variable `X` will eventually be instantiated by some recursive type `Rec(Y)S`, where `Y` <: `Z` $\vdash$ `S` <: `T`. Hence (by subsumption) the ordinary unfolding of $\mathtt{x} \in \mathtt{X}$ also has the type `[X/Z]T`.

As a final example we mention Pierce's repacking operator for existential types [Pie96]. In order to formulate the existential object encoding [PT94] in a second-order setting, one can introduce a function

```
repack  ∈
  All(X<:Some(Z)T)
   (All(Z) All(S<:T) S→S) →
   X → X
```

with the following intended meaning:

```
repack[Some(Z)S] f e  =
  open e as [Z,m] in
   pack f[Z][S]m
   as Some(Z)S.
```

Intuitively, `repack` opens its second argument, applies its first argument to it, and repackages the result. The soundness of this operation hinges on the fact that every subtype of an existential type is again an existential type.

In each of these examples one can argue operationally that the addition of the new operators is sound, in the sense that all programs of ground type (possibly containing the new operators) can be reduced to a canonical form [Car95]. In the present paper we present a more radical approach. We propose a new calculus (called $F^D_{\leq}$) in which it is literally the case that every subtype (even a variable) of a product, existential, or recursive type can be regarded as a type with the same shape. As an immediate application, the updating constructs sketched above become definable.

This is done by introducing one or more new type formers, called *type destructors*, for each type constructor that we want to equip with update operations (at present these are cartesian products, existential types, and recursive types). For example, to handle for cartesian products we introduce new type formers `.1` and `.2` (i.e., if `T` is a type, so are `T.1` and `T.2`), which extract the first and the second component of a cartesian product type. Of course, since not every type is a cartesian product, not every type of the form `T.1` is well-formed; for example, `Int.1` is not. In order to rule out these unwanted instances, we are lead to stipulate that `T.1` is well-formed only if $\mathtt{T} <: \mathtt{S}_1 \times \mathtt{S}_2$ for some types $\mathtt{S}_1$ and $\mathtt{S}_2$.

The type destructors for cartesian products obey a covariant subtyping rule: `S <: T` implies `S.i <: T.i`. In addition, we have β- and η-like type equalities:

$$\begin{array}{rcll} (\mathtt{T}_1\times\mathtt{T}_2).i & = & \mathtt{T}_i & \textsc{Beta-Prod} \\ \mathtt{T} & = & \mathtt{T.1}\times\mathtt{T.2} & \textsc{Eta-Prod} \end{array}$$

Let us see how we can define Abadi's `mix` function using these rules. If $\mathtt{X} <: \mathtt{Top}\times\mathtt{Top}$ then `X.1` and `X.2` are well-kinded and we have $\mathtt{X}=\mathtt{X.1}\times\mathtt{X.2}$, so

```
fun(X<:Top×Top) fun(e:X) fun(e':X) (e.1,e'.2)
  ∈  All(X<:Top×Top) X→X→X
```

becomes a valid typing. Notice that without type destructors the above function has the minimal type

```
All(X<:Top×Top) X→X→(Top×Top),
```

which wastes information.

To be able to write interesting (and sound) update functions, we need a type constructor that is invariant in the subtype relation. To this end, we introduce an updatable variant of the cartesian product, written $!\mathtt{T}_1\times\mathtt{T}_2$, which is invariant in its first position and covariant in the second. Here we only need the type destructor `.2`, and the corresponding η-like rule takes the form

$$\mathtt{T} = !\mathtt{S}_1\times\mathtt{T.2} \qquad \textsc{Eta-Prod-Upd}$$

whenever $\mathtt{T} <: !\mathtt{S}_1\times\mathtt{S}_2$.

The type destructor together with equation Eta-Prod-Upd allows us to define the following polymorphic update function for updatable products:

```
    fun(A<:Top) fun(X<:!A×Top)
     fun(e:X) fun(a:A)
     (a,e.2)
∈  All(A<:Top) All(X<:!A×Top) X→A→X.
```

Again, without type destructors the minimal type of this function would be

```
All(A<:Top) All(X<:!A×Top) X→A→(A×Top),
```

which represents a loss of information. This example shows how updatable products provide a way of replacing a component of a compound object by a new value. In Section 4.1 we show how this can be used to encode records with updatable fields and single field update.

1.1 An Ideal System of Type Destructors

Starting from these considerations, we have experimented with a system for type destructors which extends $F_{\leq}$ with a kinding judgement $\Gamma \vdash \mathtt{T} \in *$ to mean that `T` is a well-formed type in context Γ. Kinding rules for type destructors have subtyping premises; apart from the rules for product types introduced informally above, we have rules for type destructors corresponding to bounded existentials:

$$\frac{\Gamma \vdash \mathtt{S} <: \mathtt{Some(X<:T_1)T_2}}{\Gamma \vdash \mathtt{EBound(S)} \in *}$$

$$\frac{\Gamma \vdash \mathtt{S} <: \mathtt{Some(X<:T_1)T_2} \qquad \Gamma \vdash \mathtt{U} \in *}{\Gamma \vdash \mathtt{EBody(U,S)} \in *}$$

In addition, we have β- and η-like equalities for products (as above) and existentials

$$\begin{array}{rcl} \mathtt{EBound(Some(X<:T_1)T_2)} & = & \mathtt{T_1} \\ \mathtt{EBody(U,Some(X<:T_1)T_2)} & = & \mathtt{[U/X]T_2} \\ \mathtt{S} & = & \mathtt{Some(X<:EBound(S))} \\ & & \quad \mathtt{EBody(X,S)} \end{array}$$

(provided the right-hand side is well formed in the final rule). Similar rules can be given for other type formers, including recursive types and even universal and arrow types.

This system looks quite natural and handles all of the above examples. Alas, although the system appears to be sound, its formal metatheory has proved totally unmanageable! The most prominent defect we have found is that β-reduction on well-formed types need not terminate. To see this, suppose that $\mathtt{A} = \mathtt{Some(X<:Top)Top}$ and $\mathtt{B} = \mathtt{EBody(Z,Z)}$. The above rules yield $\mathtt{Z<:A} \vdash \mathtt{B} \in *$. Now let $\mathtt{C} = \mathtt{Some(Z<:A)B}$. The type expression `EBody(C,C)` is well-formed, but admits an infinite sequence of β-reductions.

This problem makes it very difficult if not impossible to design a complete syntax-directed presentation of subtyping in the style of $F_{\leq}$. One could now accept this lack and look for sound but incomplete semi-algorithms for subtyping and type checking. One would then have to give empirical evidence that these algorithms terminate on many interesting inputs. Indeed, preliminary experiments with the implementation suggest that this might be the case.

1.2 The System $F^D_{\leq}$

In this paper we take, however, a different approach and describe a restricted system, $F^D_{\leq}$, which does have desirable metatheoretic properties such as decidability of all judgements and type soundness. The most prominent difference between $F^D_{\leq}$ and the ideal system is the restriction to *unbounded* existential types. In this way, every type can be reduced to a normal form. Other more technical differences are that we have separated well-formedness from subtyping using a more refined kinding system and that we consider a β-redex like `(T`$_1$`×T`$_2$`).1` as definitionally equal to `T`$_1$. Finally, we forbid eta-conversion of types in certain positions, notably bounds of universal quantifiers. This simplifies the metatheory and does not seem to restrict the applicability in an essential way. However, we are confident that with some extra work this restriction could be relaxed.

The system $F^D_{\leq}$ contains a destructor for recursive types, and thus allows us to define all of the update operations mentioned above. (We show how to treat the example of structural unfolding for recursive types in Section 3.)

1.3 Translation into $F^\omega_{\leq}$

The fragment of $F^D_{\leq}$ without recursive types admits a translation into the higher-order system $F^\omega_{\leq}$ [Car90, CL91, PT94, HP95b, PS94, Com94] which is the identity on untyped terms. The details of this translation are a bit heavy notationally, but the basic idea is easy to explain and might improve the reader's intuition for type destructors. Roughly, we translate a variable binding `Z <: T` into a sequence of type variable and type operator variable bindings, followed by a `let`-binding defining `Z` in terms of these newly bound variables. For example, a binding

$$\texttt{Z <: Some(X) !X}\times\texttt{X}\times\texttt{(X}\to\texttt{Int)}$$

becomes

$$\begin{array}{l}\texttt{Z}_{\texttt{EBody(Z).2.1}} \texttt{ <: Fun(X)X} \\ \texttt{Z}_{\texttt{EBody(Z).2.2}} \texttt{ <: Fun(X)X}\to\texttt{Int} \\ \texttt{Z = Some(X) !X}\times\texttt{Z}_{\texttt{EBody(Z).2.1}}\texttt{(X)}\times\texttt{Z}_{\texttt{EBody(Z).2.2}}\texttt{(X)}.\end{array}$$

Then, for example, the $F^D_{\leq}$-type `EBody(U,Z).2.1` can be *defined* as $\texttt{Z}_{\texttt{EBody(Z).2.1}}$`(U)`. Note the similarity between the result of the translation and the original "simple existential encoding" of objects in $F^\omega_{\leq}$[PT94, HP95b].

We can thus view (the non-recursive fragment of) $F^D_{\leq}$ as a high-level syntax for such explicit type and operator quantifications. Our experience with a prototype implementation suggests that the use of $F^D_{\leq}$ instead of these explicit quantifications leads to substantially simpler and more readable code.

Extending this translation to recursive types with monotone subtyping would require an extension of $F^\omega_{\leq}$ with monotone operator subtyping (cf. [Car90, Ste97]). The ideal system with full bounded existentials does not seem to admit a translation of this kind.

Outline

Section 2 begins the formal treatment of $F^D_{\leq}$ with a definition of its syntax. Section 3 defines the kinding, eta-conversion, subtyping, and typing relations, and summarizes important meta-theoretic properties. In Section 4 we reformulate two familiar encodings of objects—the standard recursive-records model and the simple existential model—in $F^D_{\leq}$. Section 5 sketches a possible denotational semantics for the system. Section 6 offers concluding remarks and some ideas for future work. The detailed development of the metatheory of $F^D_{\leq}$ is omitted here, for the sake of brevity; it can be found in an accompanying technical report [HP97].

2 Syntax

For technical convenience, we split the syntactic class of types into two parts: *neutral types*, consisting of a type variable possibly embedded in a sequence of destructors, and *active types*, which have a concrete type constructor at the head.

			types
`T`	::=	`N`	neutral type
		`A`	active type
			neutral types
`N`	::=	`X`	type variable
		`N.1`	first projection
		`N.2`	second projection
		`EBody(T,N)`	body of an existential type
		`RBody(T,N)`	body of a recursive type
			active types
`A`	::=	`Top`	maximal type
		`Int`	constant type of integers
		$\texttt{T}_1\to\texttt{T}_2$	function type
		$\texttt{T}_1\times\texttt{T}_2$	product type
		$\texttt{!T}_1\times\texttt{T}_2$	updatable product type
		$\texttt{All(X<:T}_1\texttt{)T}_2$	universal type
		`Some(X)T`	existential type
		`Rec(X)T`	recursive type

The destructors `.1` and `.2` correspond to the product type constructor × (and, in the case of `.2`, also to the updatable product constructor `!...×`). The destructor `EBody` corresponds to the constructor `Some`; intuitively, `EBody(T,N)` can be read as "The type formed by instantiating the body of the existential type `N` with the value `T` for the bound variable." (For example, consider the type expression `EBody(Int,X)`; if `X` is later instantiated with `Some(Y)Y×Y`, then `EBody(Int,X)` will become equivalent to `Int×Int`; the definitions of substitution below and the eta-conversion relation in Section 3.2 will make this point clearer.) Similarly, the destructor `RBody` corresponds to the constructor `Rec`.

Notice that we do not provide destructors for all of the constructors. Introducing destructors for contravariant constructors such as → and `All` would give rise to destructors with contravariant subtyping behavior, which raise difficult metatheoretic problems (requiring backtracking during subtype-checking, etc.). For example, if we were to introduce destructors `Dom` and `Cod` such that $\texttt{T} =_\eta \texttt{Dom(T)}\to\texttt{Cod(T)}$ whenever $\texttt{T<:S}_1\to\texttt{S}_2$ then `Dom` would be contravariant and a goal `X.1 <: Dom(Y)?` could be solved by replacing either `X` or `Y` by its upper bound. On the other hand, we could not think of any useful applications for such contravariant destructors, so we decided to omit them.

Also, notice that—as explained in the introduction—existential types are unbounded in the present system. This is a real restriction: many object encodings can be carried out using only unbounded existentials, but some of the most interesting encodings (e.g. Abadi, Cardelli, and Viswanathan's [ACV96]) do require bounded existential types. Therefore, future research should concentrate on removing this restriction (cf. Section 6).

We will only use the `.1` destructor for non-updatable products: for updatable products it is not needed (if we know that `T <: !U×V`, then we know that the first component of `T` is *exactly* `U`), and allowing it clutters the formal development.

A *typing context* Γ is a list of bindings of the form `x:T`, `X<:T`, or `X:*` such that, whenever $\Gamma = \Gamma'$, `x:T`, Γ'' or $\Gamma = \Gamma'$, `X<:T`, Γ'', all free variables of `T` are bound in Γ'.

			contexts
Γ	::=	$\bullet$	empty context
		Γ, `x:T`	variable binding
		Γ, `X:*`	parameter binding
		Γ, `X<:T`	bounded type variable binding

If `X<:T` occurs in Γ then $\Gamma(\mathtt{X}) \overset{\text{def}}{=} \mathtt{T}$; if `X:*` occurs in Γ then $\Gamma(\mathtt{X}) \overset{\text{def}}{=} *$. A type variable whose binding has the latter form is called a *parameter*.

2.1 Definition [Substitution]: Since we restrict the application of destructors to neutral types, excluding expressions like $(\mathtt{T}_1\times\mathtt{T}_2)\mathtt{.1}$, we need to simplify type expressions when we perform a substitution. To do this we define the substitution `[V/X](T)` of type `V` for `X` in `T` by:

$$
\begin{array}{l}
[\mathtt{V/X}](\mathtt{N.i}) = \\
\quad\left\{\begin{array}{ll}
([\mathtt{V/X}]\mathtt{N})\mathtt{.i} & \text{if } [\mathtt{V/X}]\mathtt{N} \text{ neutral} \\
\mathtt{S_i} & \text{if } [\mathtt{V/X}]\mathtt{N} = \mathtt{S}_1\times\mathtt{S}_2 \\
 & \text{or (when i=2) } \mathtt{!S}_1\times\mathtt{S}_2 \\
\text{undefined} & \text{otherwise}
\end{array}\right. \\
[\mathtt{V/X}](\mathtt{EBody(T,N)}) = \\
\quad\left\{\begin{array}{ll}
\mathtt{EBody}([\mathtt{V/X}]\mathtt{T},[\mathtt{V/X}]\mathtt{N}) & \text{if } [\mathtt{V/X}]\mathtt{N} \text{ neutral} \\
[[\mathtt{V/X}]\mathtt{T/Y}]\mathtt{S} & \text{if } [\mathtt{V/X}]\mathtt{N} = \mathtt{Some(Y)S} \\
\text{undefined} & \text{otherwise.}
\end{array}\right. \\
[\mathtt{V/X}](\mathtt{RBody(T,N)}) = \\
\quad\left\{\begin{array}{ll}
\mathtt{RBody}([\mathtt{V/X}]\mathtt{T},[\mathtt{V/X}]\mathtt{N}) & \text{if } [\mathtt{V/X}]\mathtt{N} \text{ neutral} \\
[[\mathtt{V/X}]\mathtt{T/Y}]\mathtt{S} & \text{if } [\mathtt{V/X}]\mathtt{N} = \mathtt{Rec(Y)S} \\
\text{undefined} & \text{otherwise.}
\end{array}\right.
\end{array}
$$

For the other type formers, substitution is defined as usual.

As a notational convenience, the destructors are extended to active types by substitution, e.g.,

$$
\begin{array}{rcl}
\mathtt{RBody(S,Rec(X)T)} & = & [\mathtt{Rec(X)T/Y}]\mathtt{RBody(S,Y)} \\
 & = & [\mathtt{S/X}]\mathtt{T}.
\end{array}
$$

Obviously, these expressions may be undefined.

The term formers of $F^D_\leq$ are precisely the familiar ones for the type constructors listed above. Note that there are no extra syntactic forms corresponding to the type destructors.

			terms
e	::=	`i`	integer constant
		`x`	variable
		`fun(x:T)e`	function
		$\mathtt{e}_1\ \mathtt{e}_2$	application
		$\mathtt{e}_1\mathtt{,e}_2$	pair
		$\mathtt{!e}_1\mathtt{,e}_2$	updatable pair
		`e.1`	first projection
		`e.2`	second projection
		$\mathtt{fun(X<:T}_1\mathtt{)e}$	polymorphic abstraction
		`e[T]`	polymorphic application
		`fold [R]`	fold a recursive type
		`unfold [R]`	unfold a recursive type
		`pack [S,e] as T`	existential package
		`open e as [X,x] in e`	use of a package

Beta reduction on raw terms is defined as usual as the least reflexive transitive relation $\longrightarrow$ compatible with the term forming operations and closed under the following basic reduction steps:

$$
\begin{array}{lcl}
\mathtt{(fun(x:T)e)\ e'} & \longrightarrow & [\mathtt{e'/x}]\mathtt{e} \\
\mathtt{(fun(X<:T)e)[S]} & \longrightarrow & [\mathtt{S/X}]\mathtt{e} \\
([\mathtt{!}]\mathtt{e}_1\mathtt{,e}_2)\mathtt{.}i & \longrightarrow & \mathtt{e}_i \\
\mathtt{unfold[R_1]\ (fold[R_2]\ e)} & \longrightarrow & \mathtt{e} \\
\mathtt{open\ (pack[S,e]\ as\ T)} & \longrightarrow & [\mathtt{S/X}][\mathtt{e/x}]\mathtt{e'} \\
\quad\mathtt{as\ [X,x]\ in\ e'} & &
\end{array}
$$

We sometimes write $\diamondsuit$ to stand for any of the binary type constructors $!\ldots\times$, $\times$, and $\rightarrow$.

3 Typing Rules

3.1 Kinding

In order to control the applicability of type destructors we introduce a kinding relation which associates each well-formed type with a type of a special form (called a *kind*) that describes further applicability of destructors. The set of kinds is defined by the following grammar:

$$
\begin{array}{lcl}
\mathtt{K} & ::= & \mathtt{Top} \\
 & & \mathtt{X} \\
 & & \mathtt{K}_1\times\mathtt{K}_2 \\
 & & \mathtt{!T}_1\times\mathtt{K}_2 \\
 & & \mathtt{Some(X)K} \\
 & & \mathtt{Rec(X)K}
\end{array}
$$

To state the kinding relation, we need a variant of the substitution operation that treats variables differently depending on where they occur. Suppose that `K` and `L` are kinds, `S` is a type, and `X` is a parameter. Then the substitution of `S` and `K` for `X` in `L` is defined as follows:

$$
\begin{array}{lcll}
[\mathtt{S,K/X}]\mathtt{Top} & = & \mathtt{Top} & \\
[\mathtt{S,K/X}]\mathtt{X} & = & \mathtt{K} & \\
[\mathtt{S,K/X}]\mathtt{L}_1\times\mathtt{L}_2 & = & [\mathtt{S,K/X}]\mathtt{L}_1\times[\mathtt{S,K/X}]\mathtt{L}_2 & \\
[\mathtt{S,K/X}]\mathtt{!T}_1\times\mathtt{L}_2 & = & [\mathtt{S/X}]\mathtt{!T}_1\times[\mathtt{S,K/X}]\mathtt{L}_2 & \\
[\mathtt{S,K/X}]\mathtt{Some(Y)L}_2 & = & \mathtt{Some(Y)}[\mathtt{S,K/X}]\mathtt{L}_2 & \text{if } \mathtt{X}\neq\mathtt{Y} \\
[\mathtt{S,K/X}]\mathtt{Rec(Y)L}_2 & = & \mathtt{Rec(Y)}[\mathtt{S,K/X}]\mathtt{L}_2 & \text{if } \mathtt{X}\neq\mathtt{Y}
\end{array}
$$

Intuitively, `[S,K/X]L` is obtained from `L` by replacing every occurrence of `X` in `L` within a left-hand side of an updatable product by `S`, and every other occurrence by `K`. The kinding relation $\Gamma \vdash \mathtt{T} \ll \mathtt{K}$ now associates each well-formed type expression `T` with an active type `K` from which the applicability of destructors can be read off.

$$\frac{}{\Gamma \vdash \mathtt{Top} \ll \mathtt{Top}} \qquad \text{(K-Top)}$$

$$\frac{}{\Gamma \vdash \mathtt{Int} \ll \mathtt{Top}} \qquad \text{(K-Base)}$$

$$\frac{\Gamma \vdash \mathtt{T}_1 \ll \mathtt{K}_1 \qquad \Gamma \vdash \mathtt{T}_2 \ll \mathtt{K}_2}{\Gamma \vdash \mathtt{T}_1\rightarrow\mathtt{T}_2 \ll \mathtt{Top}} \qquad \text{(K-Arr)}$$

$$\frac{\Gamma \vdash \mathtt{T}_1 \ll \mathtt{K}_1 \qquad \Gamma \vdash \mathtt{T}_2 \ll \mathtt{K}_2}{\Gamma \vdash \mathtt{T}_1\times\mathtt{T}_2 \ll \mathtt{K}_1\times\mathtt{K}_2} \qquad \text{(K-Prod)}$$

$$\frac{\Gamma \vdash \mathtt{T}_1 \ll \mathtt{K}_1 \qquad \Gamma \vdash \mathtt{T}_2 \ll \mathtt{K}_2}{\Gamma \vdash \mathtt{!T}_1\times\mathtt{T}_2 \ll \mathtt{!T}_1\times\mathtt{K}_2} \qquad \text{(K-Upd)}$$

$$\frac{\Gamma,\texttt{X:*} \vdash \texttt{T}_2 \ll \texttt{K}_2}{\Gamma \vdash \texttt{Some(X)T}_2 \ll \texttt{Some(X)K}_2} \quad \text{(K-Some)}$$

$$\frac{\Gamma \vdash \texttt{T}_1 \ll \texttt{K}_1 \qquad \Gamma,\texttt{X<:T}_1 \vdash \texttt{T}_2 \ll \texttt{K}_2}{\Gamma \vdash \texttt{All(X<:T}_1\texttt{)T}_2 \ll \texttt{Top}} \quad \text{(K-All)}$$

$$\frac{\Gamma,\texttt{X:*} \vdash \texttt{T} \ll \texttt{K}}{\Gamma \vdash \texttt{Rec(X)T} \ll \texttt{Rec(X)K}} \quad \text{(K-Rec)}$$

$$\frac{\Gamma \vdash \Gamma(\texttt{X}) = \texttt{*}}{\Gamma \vdash \texttt{X} \ll \texttt{X}} \quad \text{(K-Param)}$$

$$\frac{\Gamma \vdash \Gamma(\texttt{X}) \ll \texttt{K}}{\Gamma \vdash \texttt{X} \ll \texttt{K}} \quad \text{(K-Var)}$$

$$\frac{\Gamma \vdash \texttt{N} \ll \texttt{K}_1\times\texttt{K}_2}{\Gamma \vdash \texttt{N.1} \ll \texttt{K}_1} \quad \text{(K-Fst)}$$

$$\frac{\Gamma \vdash \texttt{N} \ll \texttt{K}_1\times\texttt{K}_2}{\Gamma \vdash \texttt{N.2} \ll \texttt{K}_2} \quad \text{(K-Snd)}$$

$$\frac{\Gamma \vdash \texttt{N} \ll \texttt{!T}_1\times\texttt{K}_2}{\Gamma \vdash \texttt{N.2} \ll \texttt{K}_2} \quad \text{(K-Snd-Upd)}$$

$$\frac{\Gamma \vdash \texttt{T} \ll \texttt{K}_1 \qquad \Gamma \vdash \texttt{N} \ll \texttt{Some(X)K}_2}{\Gamma \vdash \texttt{EBody(T,N)} \ll [\texttt{T},\texttt{K}_1/\texttt{X}]\texttt{K}_2} \quad \text{(K-EBody)}$$

$$\frac{\Gamma \vdash \texttt{T} \ll \texttt{K}_1 \qquad \Gamma \vdash \texttt{N} \ll \texttt{Rec(X)K}_2}{\Gamma \vdash \texttt{RBody(T,N)} \ll [\texttt{T},\texttt{K}_1/\texttt{X}]\texttt{K}_2} \quad \text{(K-RBody)}$$

We say that T is well-kinded under Γ and write $\Gamma \vdash \texttt{T} \in \texttt{*}$ if $\Gamma \vdash \texttt{T} \ll \texttt{K}$ for some K. The most interesting rules are K-Upd, K-EBody, and K-RBody. K-EBody, for example, can be read as follows: "If N is bounded by an existential type of the form $\texttt{Some(X)K}_2$ and T is well-kinded, then the destructor application EBody(T,N) is well-kinded and has the form $\texttt{K}_2$, with $\texttt{K}_1$ (or T in left-hand sides of updatable products) substituted for the bound variable X."

Note that kinding is a (partial) function: If $\Gamma \vdash \texttt{S} \ll \texttt{K}$ and $\Gamma \vdash \texttt{S} \ll \texttt{L}$, then $\texttt{K} = \texttt{L}$.

3.2 Eta-Conversion

The *eta-conversion* relation between types, written $\Gamma \vdash \texttt{S} =_\eta \texttt{T}$, is the least equivalence relation closed under the following rules,

$$\frac{\Gamma \vdash \texttt{N} \ll \texttt{K}_1\times\texttt{K}_2}{\Gamma \vdash \texttt{N} =_\eta \texttt{N.1}\times\texttt{N.2}} \quad \text{(Eta-Prod)}$$

$$\frac{\Gamma \vdash \texttt{N} \ll \texttt{!T}\times\texttt{K}}{\Gamma \vdash \texttt{N} =_\eta \texttt{!T}\times\texttt{N.2}} \quad \text{(Eta-Upd)}$$

$$\frac{\Gamma \vdash \texttt{N} \ll \texttt{Some(X)K}}{\Gamma \vdash \texttt{N} =_\eta \texttt{Some(X)EBody(X,N)}} \quad \text{(Eta-Some)}$$

$$\frac{\Gamma \vdash \texttt{N} \ll \texttt{Rec(X)K}}{\Gamma \vdash \texttt{N} =_\eta \texttt{Rec(X)RBody(X,N)}} \quad \text{(Eta-Rec)}$$

plus congruence rules for all the type formers except the bounds of univerisal quantifiers and the first (substitutive) arguments of EBody and RBody. (The prohibition of eta-conversion in these positions is a somewhat ad-hoc restriction, needed in our proof of completeness of the syntax-directed presentation of eta-conversion and subtyping [HP97]. We conjecture that it can be relaxed.)

3.2.1 Lemma [Eta-congruence preserves kinding]: If $\Gamma \vdash \texttt{S} =_\eta \texttt{T}$ and $\Gamma \vdash \texttt{S} \ll \texttt{K}$, then $\Gamma \vdash \texttt{T} \ll \texttt{L}$ and $\Gamma \vdash \texttt{K} =_\eta \texttt{L}$.

(One might expect that, with the restricted definition of eta-conversion that we're using at the moment, this property could be made even stronger: $\texttt{K} = \texttt{L}$. But this is still not the case, for example, when $\texttt{S} = \texttt{!(X.1}\times\texttt{X.2)}\times\texttt{Top}$ and $\texttt{T} = \texttt{!X}\times\texttt{Top}$.)

3.3 Subtyping

The subtyping rules for the active type formers are the same as in (the Kernel Fun variant of) $F_\leq$; that is to say, ordinary products are covariant in both arguments, function spaces are contravariant in the first position and covariant in the second, universal quantifiers and updatable products are invariant in the first position and contravariant in the second, unbounded existentials are covariant, and recursive types obey the monotone subtyping rule mentioned in the introduction.

$$\frac{\Gamma \vdash \texttt{S} \in \texttt{*}}{\Gamma \vdash \texttt{S <: Top}} \quad \text{(S-Top)}$$

$$\Gamma \vdash \texttt{X <: } \Gamma(\texttt{X}) \quad \text{(S-Var)}$$

$$\frac{\Gamma \vdash \texttt{T}_1 \texttt{ <: } \texttt{S}_1 \qquad \Gamma \vdash \texttt{S}_2 \texttt{ <: } \texttt{T}_2}{\Gamma \vdash \texttt{S}_1\texttt{->}\texttt{S}_2 \texttt{ <: } \texttt{T}_1\texttt{->}\texttt{T}_2} \quad \text{(S-Arrow)}$$

$$\frac{\Gamma,\texttt{X<:U}_1 \vdash \texttt{S}_2 \texttt{ <: } \texttt{T}_2}{\Gamma \vdash \texttt{All(X<:U1)S}_2 \texttt{ <: All(X<:U1)T}_2} \quad \text{(S-All)}$$

$$\frac{\Gamma,\texttt{X:*} \vdash \texttt{S <: T}}{\Gamma \vdash \texttt{Some(X)S <: Some(X)T}} \quad \text{(S-Some)}$$

$$\frac{\Gamma,\texttt{Y}\in\texttt{*},\texttt{X<:Y} \vdash \texttt{S <: T}}{\Gamma \vdash \texttt{Rec(X)S <: Rec(Y)T}} \quad \text{(S-Rec)}$$

$$\frac{\Gamma \vdash \texttt{S}_1 \texttt{ <: } \texttt{T}_1 \qquad \Gamma \vdash \texttt{S}_2 \texttt{ <: } \texttt{T}_2}{\Gamma \vdash \texttt{S}_1\times\texttt{S}_2 \texttt{ <: } \texttt{T}_1\times\texttt{T}_2} \quad \text{(S-Prod)}$$

$$\frac{\Gamma \vdash \texttt{T}_2 \texttt{ <: } \texttt{T}_3 \qquad \Gamma \vdash \texttt{T}_1 \in \texttt{*}}{\Gamma \vdash \texttt{!T}_1\times\texttt{T}_2 \texttt{ <: !T}_1\times\texttt{T}_3} \quad \text{(S-Upd)}$$

The subtyping rules for the type destructors are reminiscent of the generation lemma for their active counterparts: .1 and .2 are covariant, while EBody and RBody are invariant in their first (substitutive) arguments and covariant in their second arguments. Finally, subtyping extends η-equality.

$$\frac{\Gamma \vdash \texttt{S <: T} \qquad \Gamma \vdash \texttt{S.i} \in \texttt{*} \qquad \Gamma \vdash \texttt{T.i} \in \texttt{*}}{\Gamma \vdash \texttt{S.i <: T.i}} \quad \text{(S-Proj)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \texttt{S} \ll \texttt{Some(X)K}_1 \qquad \Gamma \vdash \texttt{T} \ll \texttt{Some(X)K}_2 \\ \Gamma \vdash \texttt{U} \in \texttt{*} \qquad \Gamma \vdash \texttt{S <: T}\end{array}}{\Gamma \vdash \texttt{EBody(U,S) <: EBody(U,T)}} \quad \text{(S-EBody)}$$

$$\frac{\Gamma \vdash \texttt{S} \ll \texttt{Rec(X)S}_1 \qquad \Gamma \vdash \texttt{T} \ll \texttt{Rec(X)T}_1 \qquad \Gamma \vdash \texttt{U} \in * \qquad \Gamma \vdash \texttt{S <: T}}{\Gamma \vdash \texttt{RBody(U,S) <: RBody(U,T)}} \quad \text{(S-RBody)}$$

$$\frac{\Gamma \vdash \texttt{S} =_\eta \texttt{T} \qquad \Gamma \vdash \texttt{S} \in *}{\Gamma \vdash \texttt{S <: T}} \quad \text{(S-Conv)}$$

Notice that the destructors occurring in these rules may be defined ones, so, for example, the following is a valid derivation:

$$\frac{\dfrac{}{\texttt{Y}\in *,\ \texttt{X<:}\,\texttt{Y}\times\texttt{Y} \vdash \texttt{X <: Y}\times\texttt{Y}}\ \text{(S-Var)}}{\texttt{Y}\in *,\ \texttt{X<:}\,\texttt{Y}\times\texttt{Y} \vdash \texttt{X.1 <: Y}} \ \text{(S-Fst)}$$

Also, note that we might wish for stronger versions of some of these rules, e.g.,

$$\frac{\Gamma \vdash \texttt{U <: V} \qquad \Gamma \vdash \texttt{M <: N}}{\Gamma \vdash \texttt{RBody(U,M) <: RBody(V,N)}}$$

but this would be hard to implement as it introduces a disjunctive case in the subtyping algorithm [HP97].

3.4 Typing

At the level of typing, $F^D_{\leq}$ is standard. For example, we have the usual rule for forming existential packages (since our existentials are unbounded, we extend the context with the parameter binding `X:*`):

$$\frac{\Gamma \vdash \texttt{e} \in \texttt{Some(X)T} \qquad \Gamma, \texttt{X:*}, \texttt{y}\in\texttt{T} \vdash \texttt{b} \in \texttt{B} \qquad \texttt{X} \notin FV(\texttt{B})}{\Gamma \vdash \texttt{open e as [X,y] in b} \in \texttt{B}} \quad \text{(T-Open)}$$

The corresponding rule for `pack` is:

$$\frac{\Gamma \vdash \texttt{E} =_\eta \texttt{Some(X)T} \qquad \Gamma \vdash \texttt{e} \in \texttt{[S/X]T}}{\Gamma \vdash \texttt{pack [S,e] as E} \in \texttt{E}} \quad \text{(T-Pack)}$$

The `fold` and `unfold` constructors are treated as follows:

$$\frac{\Gamma \vdash \texttt{R} =_\eta \texttt{Rec(X)T}}{\Gamma \vdash \texttt{unfold [R]} \in \texttt{R} \to \texttt{[R/X]T}} \quad \text{(T-Unfold)}$$

$$\frac{\Gamma \vdash \texttt{R} =_\eta \texttt{Rec(X)T}}{\Gamma \vdash \texttt{fold [R]} \in \texttt{[R/X]T} \to \texttt{R}} \quad \text{(T-Fold)}$$

The typing relation also includes the usual rule of subsumption:

$$\frac{\Gamma \vdash \texttt{e} \in \texttt{S} \qquad \Gamma \vdash \texttt{S <: T}}{\Gamma \vdash \texttt{e} \in \texttt{T}} \quad \text{(T-Subsumption)}$$

As an example of the use of these rules, note that Abadi and Cardelli's "structural rule" for `unfold` expressions [AC96]

$$\frac{\Gamma \vdash \texttt{e} \in \texttt{R <: Rec(X)T}}{\Gamma \vdash \texttt{unfold [R] e} \in \texttt{[R/X]T}}$$

is derivable in $F^D_{\leq}$. If `R <: Rec(X)T` then $\texttt{R} \ll \texttt{Rec(X)K}$ by Theorem 3.5.3, so $\texttt{R} =_\eta$ `Rec(X) RBody(X,R)`. Hence, if $\texttt{e} \in \texttt{R}$, then $\texttt{e} \in$ `Rec(X) RBody(X,R)` by S-Conv and T-Subsumption, so

$$\begin{array}{lll} \texttt{unfold [R] e} & \in & \texttt{RBody(R, R)} \\ & \texttt{<:} & \texttt{RBody(R, Rec(X)T)} \\ & \text{i.e.} & \texttt{[R/X]T}. \end{array}$$

3.5 Metatheory

Syntax-directed presentations of η-equality, subtyping, and typing allow us to establish the following metatheoretic properties. (The interested reader is referred to the full version [HP97] for proofs.)

3.5.1 Theorem [Substitution]: Let $\mathcal{J}$ be `S <: T` or $\texttt{S} =_\eta \texttt{T}$ or $\texttt{e} \in \texttt{T}$.

1. If $\Gamma, \texttt{X:*}, \Delta \vdash \mathcal{J}$ and $\Gamma \vdash \texttt{U} \in *$ then $\Gamma, \texttt{[U/X]}\Delta \vdash \texttt{[U/X]}\mathcal{J}$.
2. If $\Gamma, \texttt{X<:V}, \Delta \vdash \mathcal{J}$ and $\Gamma \vdash \texttt{U <: V}$ then $\Gamma, \texttt{[U/X]}\Delta \vdash \texttt{[U/X]}\mathcal{J}$.

(Notice that this theorem refers to the "β-reducing" substitution operation, and is thus far from trivial.)

3.5.2 Theorem [Decidability]: All judgements of $F^D_{\leq}$ are decidable by syntax-directed procedures.

3.5.3 Theorem [Completeness of Kinding]:

1. If $\Gamma \vdash \texttt{S <: T}_1\times\texttt{T}_2$, then $\Gamma \vdash \texttt{S} \ll \texttt{K}_1\times\texttt{K}_2$ for some $\texttt{K}_1$ and $\texttt{K}_2$.
2. If $\Gamma \vdash \texttt{S <: !T}_1\times\texttt{T}_2$, then $\Gamma \vdash \texttt{S} \ll \texttt{!T}_1{}'\times\texttt{K}_2$ for some $\texttt{T}_1{}'$ and $\texttt{K}_1$ with $\Gamma \vdash \texttt{T}_1 =_\eta \texttt{T}_1{}'$.
3. If $\Gamma \vdash$ `S <: Some(X)T`, then $\Gamma \vdash \texttt{S} \ll$ `Some(X)K` for some `K`.
4. If $\Gamma \vdash$ `S <: Rec(X)T`, then $\Gamma \vdash \texttt{S} \ll$ `Rec(X)K` for some `K`.

This property fulfills the promise made in the introduction that every subtype of a product is a product, etc.

3.5.4 Theorem [Subject reduction]: If $\Gamma \vdash \texttt{e} \in \texttt{T}$ and $\texttt{e} \longrightarrow \texttt{e}'$ then $\Gamma \vdash \texttt{e}' \in \texttt{T}$.
(The proof of this theorem follows the same lines as in ordinary $F_{\leq}$. The crucial property is substitutivity of subtyping and kinding (Theorem 3.5.1).)

As in $F_{\leq}$, obviously wrong expressions like `((x,y) z)` or `(fun(x:T)x).2` cannot be typed using the typing rules. Subject reduction implies that such wrong expressions can never arise during evaluation of a well-typed term.

4 Examples

We now show how to extend the simple examples discussed so far to full-scale object encodings. We treat both of the well-known "simple encodings" of objects (cf. [BCP97])—one using recursive types to hide the types of instance variables and one using existential types. All the examples have been mechanically checked by our prototype implementation.

4.1 Record Syntax

To make the examples easier to read, we extend the system $F^D_{\leq}$ with conventional record notation. We introduce the following new syntactic forms:

$$\begin{array}{lll} \texttt{A} & ::= & \dots \\ & & \texttt{\{[!]l}_i\texttt{:T}_i\texttt{; ...\}} \quad \text{record type} \end{array}$$

```
e  ::=  ...
        {[!]l_i=e_i; ...}   record value
        e.l                 project ordinary field
        e..l                project updatable field
        e_1 with l:=e_2     update updatable field
```

Each field in a record is either updatable or non-updatable. For non-updatable fields, we provide the usual projection operator `e.l`. For updatable fields, we provide both projection (written `e..l`, since its encoding below is different than that of ordinary projection) and update: if `r` is a record with a updatable field `l` and `v` is a value of the appropriate type, then `r with l:=v` denotes a new record that coincides with `r` except at `l`, where its value is `v`.

These syntactic forms can all be encoded in our calculus, using a slight extension of a now-standard technique due to Cardelli [Car92]. The idea is quite simple, so we explain it informally rather than writing out a translation in full.

First, we choose some enumeration of all the labels that can appear in records. Now, an ordinary record type (with only non-updatable fields) is encoded in terms of the ordinary product type and `Top` in the usual way: by sorting its fields into the order determined by the chosen enumeration, inserting instances of `Top` for labels that do not appear in the given record type, placing a `Top` at the end, and finally dropping the labels. For example, if we take labels in alphabetical order (`a`, `b`, `c`, etc.), then the record type `{b:String}` is encoded as `(Top×String×Top)`, while `{d:Int,b:String}` is encoded as `(Top×String×Top×Int×Top)`. The encodings of record values and projection follow the same lines: `{b="red"}` is encoded as `(top, ("red", top))`, where `top` is an arbitrary value; `r.b` is encoded as `r.2.1`.

An updatable field of type `T` is encoded by placing the pair `(!T×Top)` in the appropriate position, rather than just `T`. For example, the updatable record `{!b:String}` is encoded as `(Top×(!String×Top)×Top)`. Field values and projection are encoded in the obvious way: `{!b="red"}` becomes `(top, ((!"red",top), top))`, and `r..b` becomes `r.2.1.1`. Finally, `with`-expressions are encoded by building a new record from the pieces of the original: for example, the update expression `r with b:="green"` becomes `(r.1, ((!"green", r.2.1.2), r.2.2))`. It is easy to verify that these encodings satisfy the expected typing and subtyping rules.

In a future version of $F^D_{\leq}$, we would like to include a subtyping rule of the form $\mathtt{!T_1{\times}T_2} \mathrel{\mathtt{<:}} \mathtt{T_1{\times}T_2}$. This would give us a neater encoding of records, using updatable products directly for updatable fields. For example, `{d:Int,!b:String}` would become `(Top×!String×Top×Int×Top)`. This encoding is not adequate in the present system because it disallows adding updatable fields while subtyping. For example, `{!a:Int,d:Int,!b:String}` would not be a subtype of `{d:Int,!b:String}`.

4.2 Recursive Objects

We now present a simple "objects as recursive records" encoding. The idea of the encoding is standard (cf. [BCP97] for details and references). What is interesting is the way the destructor for recursive types is used to achieve "polymorphic unfolding" in a style reminiscent of [ACV96, AC96], rather than using higher-order quantification [PT94, HP95b] or matching [BPF97, AC95] to give sufficiently refined types to the message-sending operators. The whole encoding can thus be carried out in a second-order setting.

Our running example will be the usual "functional reference cell," a simple object with three methods: `get`, `set`, and `bump`. The type of cell objects under this encoding is a recursively defined record type with three fields giving the result types of the three methods. For brevity, we'll use `Cell` in this section as an abbreviation for this type:

```
Cell  =  Rec(X) {get:Int; set:Int→X; bump:X}
```

An object with this type can be created as follows:

```
val o =
  let create =
    fix [Int→Cell]
      (fun(c:Int→Cell)
        fun(s:Int)
          fold [Cell]
            {get = s;
             set = fun(i:Int) c(i);
             bump = c(succ s)})
  in
    create(0)
  :: Cell
```

That is, we build a cell object by defining a recursive function `create` that, given an integer (representing the state of the cell) returns a record of method results, where the `set` and `bump` results are calculated by calling `create` with an appropriately updated value for the state. This function is applied to the initial state `0` to create the cell object `o`. The recursive definition of `create` uses the value-level polymorphic fixed-point operator `fix`, which can be defined in terms of recursive types [AC93].

The interesting part of the example is the typing of functions that manipulate objects by sending them messages (i.e., by unfolding the outer recursive type once and projecting one of the fields). For example, the following function sends the `get` message to an arbitrary object whose type refines `Cell`:

```
val sendget =
  fun(Y<:Cell) fun(or:Y)
    (unfold [Y] or).get
  :: All(Y<:Cell) Y→Int
```

The `sendget` function can be typed without using the special features of $F^D_{\leq}$. But the analogous `sendbump` function

```
val sendbump =
  fun(Y<:Cell) fun(or:Y)
    (unfold [Y] or).bump
  :: All(Y<:Cell) Y→Y
```

uses type destructors in an essential way. Its type can be calculated as shown in Figure 1.

4.3 Existential Objects

The "simple existential" encoding of objects [PT94, HP95b, etc.] can also be formulated in $F^D_{\leq}$. Again, the presence of type destructors allows functions manipulating objects (`sendbump`, etc.) to be written in a direct and intuitive way.

For this encoding, we keep the same interface for the cell methods, but change the type of objects so that the "state component" of an object is made visible but its type is hidden with an existential quantifier:

```
Cell  =
  Some(X) !X × X→{get:Int;set:Int→X;bump:X}
```

That is, a cell object is a pair of a state of type `X` and a collection of methods mapping `X` to the result types specified

$$
\begin{array}{rcll}
\texttt{Y} & <: & \texttt{Cell} & \text{given} \\
 & = & \texttt{Rec(X) \{get:Int; set:Int}\rightarrow\texttt{X; bump:X\}} & \text{by definition} \\
\texttt{Y} & \ll & \texttt{Rec(X) K} & \text{by Theorem 3.5.3} \\
\texttt{Y} & =_\eta & \texttt{Rec(Z)RBody(Z,Y)} & \text{by E}\textsc{ta-Rec} \\
\texttt{unfold [Y]} & \in & \texttt{Y} \rightarrow \texttt{[Y/Z]RBody(Z,Y)} & \text{by T-}\textsc{Unfold} \\
 & = & \texttt{Y} \rightarrow \texttt{RBody(Y,Y)} & \text{by defn of substitution} \\
\texttt{unfold [Y] or} & \in & \texttt{RBody(Y,Y)} & \text{by application} \\
 & <: & \texttt{[Cell/W]RBody(Y,W)} & \text{by S-RB}\textsc{ody} \\
 & = & \texttt{[[Cell/W]Y/X]\{get:Int; set:Int}\rightarrow\texttt{X; bump:X\}} & \text{by defn of substitution} \\
 & = & \texttt{\{get:Int; set:Int}\rightarrow\texttt{Cell; bump:Cell\}} & \text{by defn of substitution} \\
\texttt{(unfold [Y] or).bump} & \in & \texttt{Cell} & \text{by projection} \\
\texttt{sendbump} & \in & \texttt{All(Y<:Cell) Y}\rightarrow\texttt{Y} & \text{by abstraction.}
\end{array}
$$

Figure 1: Typing of recursive `sendbump`

by `CellI`, with the type of the state existentially quantified. Note that the state component is updatable.

Functions that manipulate objects by sending them messages are slightly more complicated here than in the recursive records model (where an object simply *was* a record of the results of its methods). For example, to send the `get` message to a cell object

```
val sendget =
  fun(Y<:Cell) fun(oe:Y)
    open oe as [Z,body] in
       (body.2 body.1).get
  :: All(Y<:Cell) Y→Int
```

we must first `open` it, binding a type variable `Y` to its hidden state type and a variable `body` to its state and methods. The methods (`body.2`) are then applied to the state (`body.1`), yielding a record of results, from which the `get` component is selected. As for the recursive record encoding, the typing of `sendget` is just as in $F_{\leq}$.

To send the `bump` message, we begin as for `get`, applying the methods to the state and projecting out the `bump` component; but this yields just a fresh state (of type `Y`), not a whole object. To obtain an object, we must repackage this state with the original methods and hide the type of the state by wrapping the whole in a new existential package:

```
val sendbump =
  fun(Y<:Cell) fun(oe:Y)
    open oe as [Z,body] in
       pack [Z, !(body.2 body.1).bump, body.2]
       as Y
  :: All(Y<:Cell) Y→Y
```

To check that `sendbump` has the claimed type, calculate as follows. First, as in the previous section:

$$
\begin{array}{lcll}
\texttt{Y} & <: & \texttt{Cell} & \text{given} \\
\texttt{Y} & \ll & \texttt{Some(X) K} & \text{by Theorem 3.5.3} \\
\texttt{Y} & =_\eta & \texttt{Some(Z) EBody(Z,Y)} & \text{by E}\textsc{ta-Some}.
\end{array}
$$

So (by T-Open), in the body of the `open` expression, the bindings of `Z` and `body` are:

$$
\begin{array}{l}
\texttt{Z:*} \\
\texttt{body} \in \texttt{EBody(Z,Y)}.
\end{array}
$$

Now,

$$
\begin{array}{l}
\texttt{EBody(Z,Y)} \\
\quad <: \texttt{EBody(Z, Some(X)!X}\times\texttt{...)} \\
\quad \text{by S-EB}\textsc{ody} \\
\quad = \texttt{!Z} \times \texttt{...} \\
\quad \text{by defn of substitution} \\
\texttt{EBody(Z,Y)} \ll \texttt{!Z}' \times \texttt{K}_2 \text{ (with } \texttt{Z}' =_\eta \texttt{Z}) \\
\quad \text{by Theorem 3.5.3} \\
\texttt{EBody(Z,Y)} =_\eta \texttt{!Z}' \times \texttt{EBody(Z,Y).2} \\
\quad \text{by E}\textsc{ta-Upd} \\
\quad =_\eta \texttt{!Z} \times \texttt{EBody(Z,Y).2} \\
\quad \text{since } =_\eta \text{ is a congruence,}
\end{array}
$$

so

$$
\texttt{body.2} \in \texttt{EBody(Z,Y).2}
$$

by projection. Moreover,

$$
\begin{array}{cl}
 & \texttt{EBody(Z,Y).2} \\
<: & \texttt{EBody(Z,Cell).2} \\
 & \text{by S-EB}\textsc{ody} \text{ and S-P}\textsc{rod} \\
= & \texttt{(!Z} \times \texttt{Z}\rightarrow\texttt{\{get:Int;set:Int}\rightarrow\texttt{Z;bump:Z\}).2} \\
 & \text{by defn of substitution} \\
= & \texttt{Z}\rightarrow\texttt{\{get:Int;set:Int}\rightarrow\texttt{Z;bump:Z\}} \\
 & \text{by defn of substitution,}
\end{array}
$$

so

$$
\texttt{(body.2 body.1).bump} \in \texttt{Z}
$$

by projection. Thus,

$$
\begin{array}{l}
\texttt{!(body.2 body.1).bump, body.2} \\
\quad \in \texttt{!Z} \times \texttt{EBody(Z,Y).2} \\
\quad =_\eta \texttt{EBody(Z,Y)},
\end{array}
$$

and hence

```
pack [Z, !(body.2 body.1).bump, body.2] as Y
```

has type `Y` by T-Pack, from which the claimed typing of `sendbump` follows by abstraction.

As before, creating a cell object with appropriate behavior is straightforward. We simply pair the initial state together with a method function and wrap the two as an existential package:

```
val o =
  pack [Int,
        !0,
```

```
        fun(s:Int)
          {get = s;
           set = fun(i:Int) i;
           bump = succ s}
        ] as Cell
  :: Cell
```

Of course, not only objects but also classes can be encoded in this framework. The power of type destructors is not needed for this encoding, but (as has been remarked elsewhere [HP95a, Pol96, etc.]) the presence of updatable record types does eliminate quite a bit of distracting boilerplate (the `get` and `put` functions of [PT94]).

5 Semantics

An important strand of future development for $F^D_{\leq}$ is denotational semantics. We give here a brief sketch of our current ideas.

It appears that we can model the full system $F^D_{\leq}$ using complete uniform pers [Ama91, AC96]. For simplicity here, we omit the recursive types and use ordinary pers. Let PER stand for the set of pers (partial equivalence relations) on the natural numbers; see [HP95a] for details on interpretation of ordinary $F_{\leq}$ using pers. The set TY of denotations for types is defined inductively as follows.

1. If $R \in PER$ then $\texttt{Per}(R) \in TY$.
2. $\texttt{Top} \in TY$.
3. If $A, B \in TY$ then $A{\times}B \in TY$.
4. If $A, B \in TY$ then $\texttt{!}A{\times}B \in TY$.
5. If $F \in PER \to TY$ then $\texttt{Some}(F) \in TY$.

Note that the symbols `Top`, `Per`, $\times$, `!`$\ldots\times$, and `Some` are free constructors of the inductive definition.

The subtyping relation $\texttt{<:} \subseteq TY \times TY$ is defined as follows:

1. $A \;\texttt{<:}\; \texttt{Top}$ (always).
2. $\texttt{Per}(R) \;\texttt{<:}\; \texttt{Per}(R')$ if $R \subseteq R'$.
3. $A{\times}B \;\texttt{<:}\; A'{\times}B'$ if $A \;\texttt{<:}\; A'$ and $B \;\texttt{<:}\; B'$.
4. $\texttt{!}A{\times}B \;\texttt{<:}\; \texttt{!}A'{\times}B'$ if $A = A'$ and $B \;\texttt{<:}\; B'$.
5. $\texttt{Some}(F) \;\texttt{<:}\; \texttt{Some}(F')$ if $F(R) \;\texttt{<:}\; F'(R)$ for each $R \in PER$.

A function $\overline{} : TY \to PER$ is defined by

$$\begin{array}{lcl}
\overline{\texttt{Per}(R)} & = & R \\
\overline{\texttt{Top}} & = & Top \\
\overline{A{\times}B} & = & \overline{A} \times \overline{B} \\
\overline{\texttt{!}A{\times}B} & = & \overline{A} \times \overline{B} \\
\overline{\texttt{Some}(F)} & = & \bigsqcup_{R \in PER} \overline{F(R)}
\end{array}$$

Here Top denotes the maximal per, $\times$ denotes cartesian product of pers, and $\sqcup$ is the symmetric, transitive closure of the set-theoretic union.

Now we can interpret $F^D_{\leq}$ type expressions in an environment which maps variables to elements of TY with the understanding that parameters are always mapped to elements of the form $\texttt{Per}(R)$. The defining clauses are as follows:

$$\begin{array}{lcl}
\llbracket \texttt{X} \rrbracket\eta & = & X \\
\llbracket \texttt{Top} \rrbracket\eta & = & \texttt{Top} \\
\llbracket \texttt{Int} \rrbracket\eta & = & \texttt{Per}(Int) \\
\llbracket \texttt{T}_1{\to}\texttt{T}_2 \rrbracket\eta & = & \texttt{Per}(\overline{\llbracket \texttt{T}_1 \rrbracket\eta} \Rightarrow \overline{\llbracket \texttt{T}_2 \rrbracket\eta}) \\
\llbracket \texttt{T}_1{\times}\texttt{T}_2 \rrbracket\eta & = & \llbracket \texttt{T}_1 \rrbracket\eta{\times}\llbracket \texttt{T}_2 \rrbracket\eta \\
\llbracket \texttt{!T}_1{\times}\texttt{T}_2 \rrbracket\eta & = & \texttt{!}\llbracket \texttt{T}_1 \rrbracket\eta{\times}\llbracket \texttt{T}_2 \rrbracket\eta \\
\llbracket \texttt{All(X<:T}_1\texttt{)T}_2 \rrbracket\eta & = & \texttt{Per}(\bigcap_{A \texttt{<:} \llbracket \texttt{T}_1 \rrbracket\eta} \overline{\llbracket \texttt{T}_2 \rrbracket\eta[\texttt{X} \mapsto A]}) \\
\llbracket \texttt{Some(X)T} \rrbracket\eta & = & \texttt{Some}(\lambda R.\llbracket \texttt{T} \rrbracket\eta[\texttt{X} \mapsto \texttt{Per}(R)]) \\
\llbracket \texttt{T.1} \rrbracket\eta & = & \left\{ \begin{array}{l} A_1, \text{ if } \llbracket \texttt{T} \rrbracket\eta = A_1{\times}A_2 \\ \text{undefined, otherwise} \end{array} \right. \\
\llbracket \texttt{T.2} \rrbracket\eta & = & \left\{ \begin{array}{l} A_2, \text{ if } \llbracket \texttt{T} \rrbracket\eta = A_1{\times}A_2 \\ \quad \text{or } \llbracket \texttt{T} \rrbracket\eta = \texttt{!}A_1{\times}A_2 \\ \text{undefined otherwise} \end{array} \right. \\
\llbracket \texttt{EBody(U,T)} \rrbracket\eta & = & \left\{ \begin{array}{l} F(\overline{\llbracket \texttt{U} \rrbracket\eta}), \text{ if } \llbracket \texttt{T} \rrbracket\eta = \texttt{Some}(F) \\ \text{undefined otherwise} \end{array} \right.
\end{array}$$

Here $\Rightarrow$ denotes function spaces of pers.

This semantics is defined for well-kinded types; η-equal types receive equal meaning, and types standing in the subtype relation are mapped to semantic types standing in the `<:` relation on TY.

On the level of terms the semantics is as usual; the soundness theorem says that if $\Gamma \vdash \texttt{e} \in \texttt{T}$ and η is an appropriate environment then $\llbracket \texttt{e} \rrbracket\eta \in dom(\overline{\llbracket \texttt{T} \rrbracket\eta})$.

Notice that this semantics does not extend to the "ideal system" with bounded existentials from the introduction: If we are allowed to apply a type destructor inside the body of an existential to the bound variable, then the semantic type former `Some` would have to take a function from TY to TY rather than a function from PER to TY as argument; then, however, TY would no longer be inductively defined. It should be possible, though, to replace the inductively defined *set* TY by an appropriately defined *domain* of "semantic type expressions." The details remain to be worked out.

6 Conclusions and Further Work

We have presented a first step towards a general theory of structural subtyping and update by adding type destructors to a version of Kernel Fun with unbounded existentials. The programming examples show that type destructors yield substantially simpler and more readable encodings of object-oriented programming idioms in typed lambda calculus.

Of course, we would like to see the syntactic restrictions on $F^D_{\leq}$ relaxed, while avoiding the bad behavior of the full "ideal system." Apart from the pragmatic solution of living with sound but incomplete checkers, one might look into more refined kinding systems that would retain much of the flexibility of $F_{\leq}$ yet rule out nonterminating type expressions. One promising idea in this direction is based on the observation that, in practice, we only seem to need the type `EBody(U,N)` if `U` is a variable. For example, to type the repack operator

$$\begin{array}{ll}
\texttt{repack} \in & \texttt{All(Z<:Some(X)T)} \\
& \quad \texttt{(All(X)All(Y<:T)Y}{\to}\texttt{Y)} \to \\
& \quad \texttt{Z}{\to}\texttt{Z},
\end{array}$$

we only need the equation $\texttt{Z} =_\eta \texttt{Some(X)EBody(X,Z)}$, and destructors other than `EBody(X,Z)` do not appear in the course of checking `repack`. The same is true for all the other examples we have checked so far. Therefore, a possible solution might be a system like $F^D_{\leq}$ but with bounded existentials (hence an `EBound` destructor as well as `EBody`) and two kinds of bound variables. The variables of the first kind are allowed to be quantified existentially and to appear as first argument in `EBody` expressions. Only variables of the first kind may be substituted for a variable of the first kind. Variables of the second kind subsume the ones of the first kind and are allowed to be quantified universally, as well as substituted by arbitrary type expressions. We hope that, in this way, one could obtain a proper extension of $F_{\leq}$ which still admits syntax-directed presentations of subtyping and type checking.

Another application of the system with type destructors is as a metalanguage for designing and justifying special-purpose term formers such as the `repack` and polymorphic `unfold` operators. Once designed, these special term formers can be added to ordinary $F_{\leq}$, obtaining the benefits of structural subtyping in particular cases at little cost in terms of meta-theoretic complexity.

References

[AC93] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993. A preliminary version appeared in POPL '91 (pp. 104–118) and as DEC Systems Research Center Research Report number 62, August 1990.

[AC95] Martín Abadi and Luca Cardelli. On subtyping and matching. In *Proceedings ECOOP '95*, pages 145–167, 1995.

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[ACV96] Martín Abadi, Luca Cardelli, and Ramesh Viswanathan. An interpretation of objects and object types. In *Principles of Programming Languages*, pages 396–409, 1996.

[Ama91] Roberto M. Amadio. Recursion over realizability structures. *Information and Computation*, 90(2):55–85, 1991.

[BCP97] Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. In *Theoretical Aspects of Computer Software (TACS), Sendai, Japan*, September 1997. An earlier version was presented as an invited lecture at the Third International Workshop on Foundations of Object Oriented Languages (FOOL 3), July 1996.

[BPF97] Kim B. Bruce, Leaf Petersen, and Adrian Fiech. Subtyping is not a good "match" for object-oriented languages. In *Proceedings of ECOOP*, volume 1241 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[Car90] Luca Cardelli. Notes about $F^{\omega}_{<:}$. Unpublished manuscript, October 1990.

[Car92] Luca Cardelli. Extensible records in a pure calculus of subtyping. Research report 81, DEC Systems Research Center, January 1992. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

[Car95] Luca Cardelli. Operationally sound update. Talk at *Higher-Order Operational Techniques in Semantics (HOOTS I)*, Cambridge, England, 1995.

[CG92] Pierre-Louis Curien and Giorgio Ghelli. Coherence of subsumption: Minimum typing and type-checking in $F_{\leq}$. *Mathematical Structures in Computer Science*, 2:55–91, 1992. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

[CL91] Luca Cardelli and Giuseppe Longo. A semantic basis for Quest. *Journal of Functional Programming*, 1(4):417–458, October 1991. Preliminary version in ACM Conference on Lisp and Functional Programming, June 1990. Also available as DEC SRC Research Report 55, Feb. 1990.

[CM91] Luca Cardelli and John Mitchell. Operations on records. *Mathematical Structures in Computer Science*, 1:3–48, 1991. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994); available as DEC Systems Research Center Research Report #48, August, 1989, and in the proceedings of MFPS '89, Springer LNCS volume 442.

[CMMS94] Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of system F with subtyping. *Information and Computation*, 109(1–2):4–56, 1994. A preliminary version appeared in TACS '91 (Sendai, Japan, pp. 750–770).

[Com94] Adriana B. Compagnoni. Decidability of higher-order subtyping with intersection types. In *Computer Science Logic*, September 1994. Kazimierz, Poland. Springer *Lecture Notes in Computer Science* 933, June 1995. Also available as University of Edinburgh, LFCS technical report ECS-LFCS-94-281, titled "Subtyping in $F^{\omega}_{\wedge}$ is decidable".

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4), December 1985.

[FM96] Kathleen Fisher and John Mitchell. The development of type systems for object-oriented languages. *Theory and Practice of Object Systems*, 1(3):189–220, 1996.

[HP95a] Martin Hofmann and Benjamin Pierce. Positive subtyping. In *Proceedings of Twenty-Second Annual ACM Symposium on Principles of Programming Languages*, pages 186–197. ACM, January 1995. Full version in *Information and Computation*, volume 126, number 1, April 1996. Also available as University of Edinburgh technical report ECS-LFCS-94-303, September 1994.

[HP95b] Martin Hofmann and Benjamin Pierce. A unifying type-theoretic framework for objects. *Journal of Functional Programming*, 5(4):593–635, October 1995. Previous versions appeared in the Symposium on Theoretical Aspects of Computer Science, 1994, (pages 251–262) and, under the title "An Abstract View of Objects and Subtyping (Preliminary Report)," as University of Edinburgh, LFCS technical report ECS-LFCS-92-226, 1992.

[HP97] Martin Hofmann and Benjamin C. Pierce. Type destructors. Technical report, Computer Science Department, Indiana University, 1997. Available through `http://www.cs.indiana.edu/hyplan/pierce.html`.

[Pie96] Benjamin C. Pierce. Even simpler type-theoretic foundations for oop. Manuscript (circulated electronically), March 1996.

[Pol96] Erik Poll. Width-subtyping and polymorphic record update. Manuscript, June 1996.

[PS94] Benjamin Pierce and Martin Steffen. Higher-order subtyping. In *IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET)*, 1994. Full version in *Theoretical Computer Science*, vol. 176, no. 1–2, pp. 235–282, 1997.

[PT94] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, April 1994. A preliminary version appeared in Principles of Programming Languages, 1993, and as University of Edinburgh technical report ECS-LFCS-92-225, under the title "Object-Oriented Programming Without Recursive Types".

[Ste97] Martin Steffen. *Polarized Higher-Order Subtyping.* PhD thesis, Universität Erlangen-Nürnberg, 1997. Forthcoming.

Using modules as classes

Jérôme Vouillon*

Abstract

We show how classes can be combined with a ML-like module system, so that the same constructions can be used for both classes and modules. For that, we are brought to slightly extend the module system with implicit type quantification of module types.

Introduction

Classes and modules share a lot of properties: they offer some form of abstraction; they structure the code and facilitate its reuse. Due to this overlapping of functionalities, it is sometimes difficult in a language providing both constructions to make a choice between them. The language Objective Caml [5] possesses a rich class-based object oriented extension [7] and a powerful higher-order module system [6]. However, classes and modules are merely juxtaposed. This paper proposes a theoretical background for a system which provides only one structuring mechanism which can be used for both classes and modules. This is obtained by identifying classes to modules. Some constructs of classes are mapped to corresponding constructs in modules (for instance, class bodies are mapped to structures); other constructs of classes, such as methods, are just added to the module language. Most of these steps are easy extensions to the module language. However, we still need a small but more significant extension of the existing module systems: just as ML types are implicitly quantified, module types would also be implicitly quantified over some type variables.

This paper is organized as follows. We first give a brief introduction to Objective ML[1]. We then describe a ML-style module language. In section 3, Objective ML classes are compared to this module language, and we show how they can be combined. In section 4, we presents more formally the resulting language. In section 5, we discuss the impact of the proposed extension to modules. In section 6, we discuss type inference. In section 7, we explain what restrictions should be made to ensure soundness in presence of imperative features. In section 8, we show the limits of our integration of classes and modules. In section 9, we compare our proposal with other works.

*Author's address: INRIA-Rocquencourt, B.P. 105, F-78153 Le Chesnay Cedex, France. Email: Jerome.Vouillon@inria.fr

[1]The reader is directed to [7] for a more complete description of Objective ML.

Presented at FOOL'5, 1998

1 Classes in Objective ML: an overview

The class language presented here is Objective ML, as formalized in [7], with a few minor modifications: self is not a keyword, but is explicitly bound in each method; in signatures, method types are only listed in self type.

A simple class can be defined as a *structure* containing some instance variables and some methods.

$$\texttt{class}\ \mathit{origin} = \\ \quad \texttt{struct val}\ x = 0;\ \texttt{method}\ \mathit{getx} = \varsigma(y)\ x\ \texttt{end}$$

This class has an instance variable x initialized to 0 and a method *getx*. When this method is invoked, y is bound to self (that is, the object the method belongs to.) In this case, this argument is actually ignored, and the method returns the value 0 of the instance variable. The type inferred for this class is

$$\forall(\alpha)\ \texttt{sig}\ (\langle\, \mathit{getx} : \mathit{int};\ \alpha\,\rangle)\ \texttt{val}\ x : \mathit{int};\ \texttt{method}\ \mathit{getx}\ \texttt{end}$$

The type between parentheses is the self type, the type of an object of this class. It is composed of a sequence of method types. The *row variable* α can be instantiated in subclasses to extend the class with new methods. The other components of the signature reveal that the class contains an instance variable x and that the method *getx* is indeed defined.

Objective ML actually provides a simple class language. A class can be defined as a *function* taking an initialization parameter and returning a class value. This is demonstrated by the following class.

$$\texttt{class}\ \mathit{genpoint} = \texttt{fun}\ x_0 \to \\ \quad \texttt{struct val}\ x = x_0;\ \texttt{method}\ \mathit{getx} = \varsigma(y)\ x\ \texttt{end}$$

The type of this class is

$$\forall(\alpha)\ \forall(\alpha')\ \big(\alpha \to \texttt{sig}\ (\langle\, \mathit{getx} : \alpha;\ \alpha'\,\rangle) \\ \qquad \texttt{val}\ x : \alpha;\ \texttt{method}\ \mathit{getx} \\ \qquad \texttt{end}\big)$$

Then, the class expression $\big(\mathit{genpoint}\ 1\big)$ is a class whose instance variable x is bound to 1. The operator `new` is used to create objects. An object is composed of a copy of the instance variables and methods of its class, but only its methods can be directly accessed. This is reflected by its type. For instance, $(\texttt{new}\ genpoint\ 1)$ is an object of type $\langle\, getx : int;\ \epsilon\,\rangle$. In this type, the row variable has been instantiated to the empty row ϵ, as the object has no other method than *getx*. The class *genpoint* could just as well be applied to a float, rather than an integer : this is a polymorphic class.

Classes can be extended by *inheritance*. The component `inherit` c of a class structure includes all components of the parent structure c into the current structure. The type for self must be an instance of the parent type for self. The class *cpoint* below inherits the instance variable x and the method *getx* from its parent *genpoint*, and defines a new method *color*.

$$\begin{array}{l}\texttt{class}\ cpoint = \texttt{fun}\ (c : string) \to \\ \quad \texttt{struct inherit}\ genpoint\ 1;\ \texttt{method}\ color = c\ \texttt{end}\end{array}$$

The type of class *cpoint*, given below, reveals the inherited instance variable and both methods.

$$\begin{array}{l}\forall(\alpha)\ string \to \\ \qquad \texttt{sig}\ (\langle\, getx : int;\ color : string;\ \alpha\,\rangle) \\ \qquad\quad \texttt{val}\ x : int;\ \texttt{method}\ getx;\ \texttt{method}\ color \\ \qquad \texttt{end}\end{array}$$

A new definition of a method would override any inherited definition.

2 ML-style modules

We now present the module system that will be compared to the class language of the previous section. A simple module is composed of a *structure* containing several components: type definitions, values, sub-modules.

$$\begin{array}{l}\texttt{module}\ intOrder = \\ \quad \texttt{struct type}\ t = int;\ \texttt{val}\ less\ (x : t)\ (y : t) = x < y\ \texttt{end}\end{array}$$

A module component can be accessed using the dot notation: the function *intOrder.less* compares two objects of type *intOrder.t* which is equal to type *int*.

A module *signature* lists the types of its components.

$$\texttt{sig type}\ t = int;\ \texttt{val}\ less : t \to t \to bool\ \texttt{end}$$

A module can also be a *functor*, expecting to be applied to another module conforming to a given interface, and returning a new module.

$$\begin{array}{l}\texttt{module}\ listOrder = \\ \quad \texttt{functor}\ (ord : \texttt{sig} \\ \qquad\qquad\qquad \texttt{type}\ t;\ \texttt{val}\ less : t \to t \to bool \\ \qquad\qquad\quad \texttt{end}) \to \\ \qquad \texttt{struct} \\ \qquad\quad \texttt{type}\ t = ord.t\ list;\ \texttt{val}\ less\ (l1 : t)\ (l2 : t) = \ldots \\ \qquad \texttt{end}\end{array}$$

The signature of this functor is

$$\begin{array}{l}\texttt{functor}\ (ord : \texttt{sig} \\ \qquad\qquad\qquad \texttt{type}\ t;\ \texttt{val}\ less : t \to t \to bool \\ \qquad\qquad\quad \texttt{end}) \to \\ \quad \texttt{sig} \\ \qquad \texttt{type}\ t = ord.t\ list;\ \texttt{val}\ less : t \to t \to bool \\ \quad \texttt{end}\end{array}$$

This functor can be applied to the previous module, yielding a module of type

$$\texttt{sig type}\ t = intOrder.t\ list;\ \texttt{val}\ less : t \to t \to bool\ \texttt{end}$$

A module can *extend* another module, including the components of this module (the `open` construct has the same meaning as in SML: components from the opened module are bound in the current module).

$$\begin{array}{l}\texttt{module}\ intOrder2 = \\ \quad \texttt{struct} \\ \qquad \texttt{open}\ intOrder; \\ \qquad \texttt{val}\ min\ x\ y = \texttt{if}\ less\ x\ y\ \texttt{then}\ x\ \texttt{else}\ y \\ \quad \texttt{end}\end{array}$$

The module *intOrder2* above contains the same components as the module *intOrder*, plus the value *min*.

3 Comparison between classes and modules

A small class language and a module language was presented in the previous sections. These languages are very similar: they both have structures, abstraction and application. Figure 1 provides an informal comparison of the syntax of these two languages, emphasizing their similarities. There are, however, some differences, and the remainder of this section will show how we can get rid of them.

A first difference is that a class abstraction maps values to class values, whereas a functor maps modules to modules. Then, structures do not contain the same components. Types also differ significantly. Class types are quantified and class structure components may have non-closed types: free type variables can occur in self type and in instance variable types. On the contrary, all core types occurring in a module type must be closed (and so, module types do not need to be quantified.) Finally, class signatures contain a distinguished component holding self type.

There is however no incompatibility. This will be shown in the remaining of this section, where we will review the differences in more details.

Class abstractions can be encoded as functors: an expression $\texttt{fun}\ x \to m$ would be translated into the expression

$$\texttt{functor}\ (z : \texttt{sig val}\ x : \tau\ \texttt{end}) \to m[z.x/x]$$

There is actually a problem here, as the type τ is usually unknown. A solution will be provided latter in this section.

It seems natural to allow structures to contain both class and module components. Inheritance (`inherit`) and inclusion (`open`) are actually the same operation: indeed, they both include all components of another structure. Value components (`val`) also have a compatible semantics.

Class type quantification is a more serious problem. It is possible to encode explicit quantification in the module language [3]: $\forall(\alpha)\ m$ can be rewritten as

$$\texttt{functor}\ (z : \texttt{sig type}\ t\ \texttt{end}) \to m[z.t/\alpha]$$

However, a value will have to be explicitly provided for type t to remove the quantification. This does not mix well with a core language with implicit polymorphism such as ML. For instance, the creation of an object of class *genpoint* (presented in section 1) would have to be verbosely written:

$$genpoint\ \big(\texttt{sig type}\ t = int;\ \texttt{val}\ x = 1\ \texttt{end}\big)$$

We therefore propose to allow the implicit quantification of module types, using the following natural rules to introduce and eliminate quantification:

Class and module expressions

$c ::= z$	$m ::= p$	Access to a named value
$\mid$ `fun` $z \to c$	$\mid$ `functor` $(z : \phi) \to m$	Abstraction
$\mid c\ e$	$\mid m\ m$	Application
$\mid$ `struct` b `end`	$\mid$ `struct` b `end`	Structure
$b ::= \epsilon \mid d; b$	$b ::= \epsilon \mid d; b$	

Structure components

$d ::=$ `inherit` c	$d ::=$ `open` m	Inheritance / inclusion
$\mid$ `val` $x = e$	$\mid$ `val` $x = e$	Value
$\mid$ `method` $y = \varsigma(x)\ e$		Method
	$\mid$ `type` $t = \tau$	Type
	$\mid$ `module` $z = m$	Module
	$T ::= \tau \mid \forall(\alpha)\ T$	Type scheme
$\Phi ::= \phi \mid \forall(\alpha)\ \Phi$		Class type scheme

Class and module types

$\phi ::= \tau \to \phi$	$\phi ::=$ `functor` $(z : \phi) \to \phi$	Functional type
$\mid$ `sig` $(\tau)\ \chi$ `end`	$\mid$ `sig` χ `end`	Signature type
$\chi ::= \epsilon \mid \psi; \chi$	$\chi ::= \epsilon \mid \psi; \chi$	

Signature components

$\psi ::=$ `val` $x : \tau$	$\psi ::=$ `val` $x : T$	Value
$\mid$ `method` m		Method
	$\mid$ `type` $t = \tau$	Manifest type
	$\mid$ `type` t	Abstract type
	$\mid$ `module` $z : \phi$	Module

Figure 1: Comparison between a class language and a module language

$$\frac{A \vdash m : \forall(\alpha)\ \Phi}{A \vdash m : \Phi[\tau/\alpha]} \qquad \frac{A \vdash m : \Phi \qquad \alpha \notin \mathrm{FV}(A)}{A \vdash m : \forall(\alpha)\ \Phi}$$

It appears useful to limit the implicit generalization of type variables at the top of structures. This can be done by explicitly binding type variables:

$$\frac{A;\ \big(\texttt{var}\ \alpha = \tau\big) \vdash m : \Phi}{A \vdash \big([\alpha]\ m\big) : \Phi}$$

During the typing of the module expression m, the type variable α is bound to some type τ. Then, the free variables of τ cannot be generalized in m, as they occur in the environment. With this extension, the expression $\big(\texttt{fun}\ x \to m\big)$ can be encoded as

$$[\alpha]\ \texttt{functor}\ (z : \texttt{sig}\ \texttt{val}\ x : \alpha\ \texttt{end}) \to m[z.x/x]$$

Other differences in types are all minor: self type can be considered as just another component of a signature; the type of values is not generalized in class signatures, but simply because this is not useful.

4 Presentation of the extended module system

We now describe the full module system. The module system presented here is a variant of the one proposed by Xavier Leroy [6]. In this variant, names appearing in a program are not annotated by stamps. Stamps only appear in types, where they are really necessary to resolve possible ambiguities. The core language is left mostly unspecified.

4.1 Syntax

The syntax is presented in figure 2. Variables z, t, x and y are names (for module, type, value and method components of structures, respectively.) Names are not subject to α-conversion. Types are denoted by Greek letters. Type schemes are denoted by the capital letter associated to the small letter denoting the corresponding types. The tildes (˜) over these syntactic types differentiate them from the enriched types that will be necessary for typing.

The following additions have been made to the syntax. Types are enriched with recursive types and object types. The module expression $[\alpha]\ m$ explicitly binds the type variable α in m. The component $\texttt{method}\ y = \varsigma(x)\ e$ defines a method; during an invocation of this method, its body e is evaluated with the variable x bound to the object the method belongs to. The component $\texttt{inherit}\ m$ includes the components of the structure m into the current structure. Module types are quantified. A signature $\big(\texttt{sig}\ (\tilde\tau)\ \tilde\chi\ \texttt{end}\big)$ is composed not only of a sequence of component types $\tilde\chi$, but also contains the self type $\tilde\tau$. The signature component $\texttt{method}\ y$ indicates that the module defines the method y.

$\tilde p ::= z \mid \tilde p.z$	Access path
$e ::= x \mid \tilde p.x \mid \texttt{new}\ m$ $\quad \mid \texttt{fun}\ x \to e \mid \ldots$	Expression
$\tilde T ::= \tilde\tau \mid \forall(\alpha)\ \tilde T$	Core type scheme
$\tilde\tau ::= t \mid \tilde p.t \mid \alpha \mid \tilde\tau \to \tilde\tau$ $\quad \mid \mu(\alpha)\ \tilde\tau \mid \langle\, \tilde\omega\, \rangle$	Core type
$\tilde\omega ::= \epsilon \mid \alpha \mid y : \tilde\tau;\ \tilde\omega$	Object type
$m ::= \tilde p \mid \texttt{struct}\ b\ \texttt{end}$ $\quad \mid \texttt{functor}\ (z : \tilde\Phi) \to m$ $\quad \mid m\ m \mid (m : \tilde\Phi) \mid [\alpha]\ m$	Module expression
$b ::= \epsilon \mid d;\ b$	Module body
$d ::= \texttt{val}\ x = e$ $\quad \mid \texttt{method}\ y = \varsigma(x)\ e$ $\quad \mid \texttt{module}\ z = m$ $\quad \mid \texttt{type}\ t = \tilde\tau$ $\quad \mid \texttt{inherit}\ m$	Module component
$\tilde\Phi ::= \tilde\phi \mid \forall(\alpha)\ \tilde\Phi$	Module type scheme
$\tilde\phi ::= \texttt{functor}\ (z : \tilde\Phi) \to \tilde\Phi$ $\quad \mid \texttt{sig}\ (\tilde\tau)\ \tilde\chi\ \texttt{end}$	Module type
$\tilde\chi ::= \epsilon \mid \tilde\psi;\ \tilde\chi$	Module body type
$\tilde\psi ::= \texttt{val}\ x : \tilde T$ $\quad \mid \texttt{method}\ y$ $\quad \mid \texttt{module}\ z : \tilde\Phi$ $\quad \mid \texttt{type}\ t \mid \texttt{type}\ t = \tilde\tau$	Module component type

Figure 2: Syntax

4.2 Typing rules

More precise types, presented in figure 3, are required for typing: names may be ambiguous and are thus annotated by a stamp, forming identifiers (this is the only difference between the syntactic types previously described and the types in this figure.) Identifiers are subject to α-conversion. However, only the stamp part can be renamed: names must still remain fixed. Types are considered equal modulo renaming of their bound identifiers.

$$p ::= z_i \mid p.z \qquad \text{Type access path}$$

$$T ::= \tau \mid \forall(\alpha)\, T \qquad \text{Core type scheme}$$

$$\tau ::= t_i \mid p.t \mid \alpha \mid \tau \to \tau \mid \mu(\alpha)\,\tau \mid \langle\, \omega\,\rangle \qquad \text{Core type}$$

$$\omega ::= \epsilon \mid \alpha \mid y : \tau ; \omega \qquad \text{Object type}$$

$$\Phi ::= \phi \mid \forall(\alpha)\, \Phi \qquad \text{Module type scheme}$$

$$\phi ::= \texttt{functor}\ (z_i : \Phi) \to \Phi \mid \texttt{sig}\ (\tau)\ \chi\ \texttt{end} \qquad \text{Module type}$$

$$\chi ::= \epsilon \mid \psi ; \chi \qquad \text{Module body type}$$

$$\psi ::= \texttt{val}\ x_i : T \mid \texttt{method}\ y \mid \texttt{module}\ z_i : \Phi \mid \texttt{type}\ t_i \mid \texttt{type}\ t_i = \tau \qquad \text{Module component type}$$

Figure 3: Types

An environment is a sequence containing signature components and bindings of a type variable to a type. An identifier cannot be bound twice in an environment (*i.e.* a name cannot be bound twice with the same stamp, but it is possible to bind two identifiers with the same name and different stamps.)

$$A ::= \epsilon \mid A;\ \psi \mid A;\ \big(\texttt{var}\ \alpha_i = \tau\big)$$

The notation $A(n)$ stands for the rightmost binding of an identifier of name n in the environment A. We write $\mathrm{FV}(A)$ and $\mathrm{FV}(\tau)$ for the free type variables of respectively an environment A and a type τ. The operation $A + \chi$ discharges the binding of the module body χ inside the environment A. It is defined by $A + \big(\psi;\ \chi\big) = \big(A;\ \psi\big) + \chi$ and $A + \epsilon = A$. The operation $\phi +_A \phi'$ concatenates the module signatures ϕ and ϕ'. It is defined by $\big(\texttt{sig}\ (\tau)\ \chi\ \texttt{end}\big) +_A \big(\texttt{sig}\ (\tau')\ \chi'\ \texttt{end}\big) = \texttt{sig}\ (\tau')\ \chi + \chi'\ \texttt{end}$ when $A \vdash \tau \approx \tau'$, where the operation $\chi + \chi'$ is defined by $\big(\psi;\ \chi\big) + \chi' = \psi;\ \big(\chi + \chi'\big)$ and $\epsilon + \chi' = \chi'$.

The typing rules assign module types to module expressions ($A \vdash m : \Phi$.) They make use of a notion of type equivalence $A \vdash T \approx T'$ and a subtyping relation between module type schemes $A \vdash \Phi <: \Phi'$. Both judgments are standard (except their extension to type schemes.) They are defined in appendix A. The relation $\approx$ is extended to module types ($A \vdash \Phi \approx \Phi'$) as the equivalence relation associated to the preorder $<:$. The translation of types $A \vdash \tilde{T} : T$ and $A \vdash \tilde{\Phi} : \Phi$ (annotating names with stamps) is presented in appendix B.

So as to make the typing rules more readable, some implicit assumptions are systematically made: all types appearing in the typing rules must be well-formed, and the bound variables of module types must be disjoint from the bound variables of the environment. Well-formedness of types is defined in appendix C: all type constructors appearing free in a type or a module type must be bound in the environment; moreover, two components of the same signature cannot have the same name.

Figure 4 gives the typing rules for the core module language. These typing rules are usual. In rule Variable, the type of the module z is enriched in order to reflect that its type components come from the module identifier z_i, using the strengthening operator Φ/z_i described in appendix D. In rule Projection, the type Φ of module z_i may depend on other components of the structure χ. The type of $\tilde{p}.z$ is thus a type Φ', equivalent to Φ but not depending on these components. Other rules are straightforward.

Figure 5 recalls the typing rules for instantiation and generalization of module types previously presented in section 3. The rules Inst and Gen are the direct transposition to module types of the standard rules for instantiation and generalization of type schemes. The rule Quant prevents the generalization of type variables bound by the syntactic type variable α during the typing of the module expression m.

The typing rules for the class components of structures are contained in figure 6. Rule Inherit is mostly syntactic sugar: rule Then-2 accomplish the actual merge of two module signatures. A method is typed like a function (rule Method); the argument type of this function must be the self type τ, while its result type must be the type of method y in τ.

The typing rule for the operator `new` is adapted from Objective ML. The type of the object `new` m is the self type τ, and all its methods must be defined (rightmost premise.)

(New)
$$\frac{A \vdash m : \big(\texttt{sig}\ (\tau)\ \chi\ \texttt{end}\big) \qquad A \vdash \tau \approx \langle\, y : \tau_y\,\rangle^{(\texttt{method}\ y) \in \chi}}{A \vdash \big(\texttt{new}\ m\big) : \tau}$$

4.3 Translation from Objective ML

The language presented above contains all syntactic elements of Objective ML, except class abstraction and class application. This elements can however be considered as syntactic sugar. The translation $[\![e]\!]$ of an Objective ML expression e is defined by induction on the syntax. We only present the rules for class abstraction and application. Other rules are trivial.

$$[\![\texttt{fun}\ x \to c]\!] = [\alpha]\ [\alpha']\ \big(\texttt{functor}\ (z : \texttt{sig}\ (\alpha')\ \texttt{val}\ x : \alpha\ \texttt{end}) \to [\![c]\!][z.x/x]\big)$$

$$[\![c\ e]\!] = [\![c]\!]\ \big(\texttt{struct val}\ x = [\![e]\!]\ \texttt{end}\big)$$

5 Impact of the extension

Our extension is conservative. In particular, it preserves the expressiveness of the extended module system. We conjecture its soundness. Indeed, the addition of module type quantification should alter only locally, in an harmless way, a proof of soundness. We have not undertaken such a proof though, by lack of a simple semantics for modules. We have not been able to found a simple reduction semantics. It may actually be possible to adapt the one proposed by Judicaël Courant [2] to our language. However, this semantics is not eager. Furthermore, it does not allow to reduce inheritance: an object body would have to keep a hierarchical structure.

The extension allows to make type parameterization of functors implicit. Without the extension, abstract types

(VARIABLE)

$$\frac{A(z) = \big(\texttt{module}\ z_i : \Phi\big)}{A \vdash z : \Phi/z_i}$$

(PROJECTION)

$$\frac{A \vdash \tilde{p} : \big(\texttt{sig}\ (\tau)\ \chi\ \texttt{end}\big) \qquad \big(\texttt{module}\ z_i : \Phi\big) \in \chi \qquad A + \chi \vdash \Phi \approx \Phi'}{A \vdash \tilde{p}.z : \Phi'}$$

(STRUCTURE)

$$\frac{A \vdash b : \phi}{A \vdash \big(\texttt{struct}\ b\ \texttt{end}\big) : \phi}$$

(CONSTRAINT)

$$\frac{A \vdash m : \Phi' \qquad A \vdash \tilde{\Phi} : \Phi \qquad A \vdash \Phi' <: \Phi}{A \vdash (m : \tilde{\Phi}) : \Phi}$$

(FUNCTOR)

$$\frac{A \vdash \tilde{\Phi}' : \Phi' \qquad A;\ \big(\texttt{module}\ z_i : \Phi'\big) \vdash m : \Phi}{A \vdash \big(\texttt{functor}\ (z : \tilde{\Phi}') \to m\big) : \big(\texttt{functor}\ (z_i : \Phi') \to \Phi\big)}$$

(APP)

$$\frac{A \vdash m : \big(\texttt{functor}\ (z_i : \Phi'_1) \to \Phi_1\big) \qquad A \vdash m' : \Phi'_2 \qquad A \vdash \Phi'_2 <: \Phi'_1 \qquad A;\ \big(\texttt{module}\ z_i : \Phi'_2\big) \vdash \Phi_1 \approx \Phi_2}{A \vdash \big(m\ m'\big) : \Phi_2}$$

(EMPTY)

$$\frac{}{A \vdash \epsilon : (\texttt{sig}\ (\tau)\ \epsilon\ \texttt{end})}$$

(THEN-1)

$$\frac{A \vdash d : \psi \qquad A;\ \psi \vdash b : \texttt{sig}\ (\tau)\ \chi\ \texttt{end}}{A \vdash \big(d;\ b\big) : \big(\texttt{sig}\ (\tau)\ \psi;\ \chi\ \texttt{end}\big)}$$

(VALUE)

$$\frac{A \vdash e : T}{A \vdash \big(\texttt{val}\ x = e\big) : \big(\texttt{val}\ x_i : T\big)}$$

(MODULE)

$$\frac{A \vdash m : \Phi}{A \vdash \big(\texttt{module}\ z = m\big) : \big(\texttt{module}\ z_i : \Phi\big)}$$

(TYPE)

$$\frac{A \vdash \tilde{\tau} : \tau \qquad \mathrm{FV}(\tau) = \emptyset}{A \vdash \big(\texttt{type}\ t = \tilde{\tau}\big) : \big(\texttt{type}\ t_i = \tau\big)}$$

Figure 4: Typing rules for core structure and module expressions

(INST)

$$\frac{A \vdash m : \forall(\alpha)\ \Phi}{A \vdash m : \Phi[\tau/\alpha]}$$

(GEN)

$$\frac{A \vdash m : \Phi \qquad \alpha \notin \mathrm{FV}(A)}{A \vdash m : \forall(\alpha)\ \Phi}$$

(QUANT)

$$\frac{A;\ \big(\texttt{var}\ \alpha_i = \tau\big) \vdash m : \Phi}{A \vdash \big([\alpha]\ m\big) : \Phi}$$

Figure 5: Typing rule for module type quantification

(THEN-2)

$$\frac{A \vdash d : \phi \qquad A + \phi \vdash b : \phi'}{A \vdash \big(d;\ b\big) : \big(\phi +_A \phi'\big)}$$

(INHERIT)

$$\frac{A \vdash m : \phi}{A \vdash \big(\texttt{inherit}\ m\big) : \phi}$$

(METHOD)

$$\frac{A \vdash \big(\texttt{fun}\ x \to e\big) : \tau' \to \tau \qquad A \vdash \tau' \approx \langle\, y : \tau;\ \omega\,\rangle}{A \vdash \big(\texttt{method}\ y = \varsigma(x)\ e\big) : \big(\texttt{sig}\ (\tau')\ \texttt{method}\ y\ \texttt{end}\big)}$$

Figure 6: Typing rules for inheritance and methods

in the signature of functor arguments are used for parameterizing modules over types. This is demonstrated by the following example, the type of a functor implementing sets.

$$\begin{array}{l}
\texttt{functor}\\
\quad (\mathit{ord} : \texttt{sig type}\ u;\ \texttt{val}\ \mathit{compare} : u \to u \to \mathit{int}\ \texttt{end}) \to\\
\quad \texttt{sig}\\
\qquad \texttt{type}\ t;\ \texttt{val}\ \mathit{empty} : t;\ \texttt{val}\ \mathit{add} : t \to \mathit{ord}.u \to t;\ \ldots\\
\quad \texttt{end}
\end{array}$$

When applying a functor of this type, both the type u of the set elements and the function *compare* for comparing elements must be provided. With our extension, the type u is not required anymore: the functor can be rewritten to have the following type.

$$\begin{array}{l}
\forall(\alpha)\ \big(\texttt{functor}\\
\qquad (\mathit{ord} : \texttt{sig val}\ \mathit{compare} : \alpha \to \alpha \to \mathit{int}\ \texttt{end}) \to\\
\qquad \texttt{sig}\\
\qquad\quad \texttt{type}\ \alpha\ t;\ \texttt{val}\ \mathit{empty} : \alpha\ t;\\
\qquad\quad \texttt{val}\ \mathit{add} : \alpha\ t \to \alpha \to \alpha\ t;\ \ldots\\
\qquad \texttt{end}\big)
\end{array}$$

Then, when applying the functor, only the comparison function will have to be provided; the type of the elements will be inferred. This type can even be polymorphic, which is not possible without the extension. Indeed, the result of the application of a functor of the preceding type to a structure of type

$$\texttt{sig val}\ \mathit{compare} : \forall(\alpha)\ \alpha \to \alpha \to \mathit{int}\ \texttt{end}$$

have type

$$\begin{array}{l}
\forall(\alpha)\ \texttt{sig}\\
\qquad \texttt{type}\ \alpha\ t;\ \texttt{val}\ \mathit{empty} : \alpha\ t;\\
\qquad \texttt{val}\ \mathit{add} : \alpha\ t \to \alpha \to \alpha\ t;\ \ldots\\
\quad \texttt{end}
\end{array}$$

which is equivalent to type

```
sig
   type α t; val empty : ∀(α) α t;
   val add : ∀(α) α t → α → α t; ...
end
```

In the second functor type, the type t has a type parameter, which is constrained to be the quantification variable α[2]. It is indeed necessary to make it explicit that the type t depends on the variable α: For instance, let us assume that our sets have a function *max* that returns the greatest element of the set. In the last module type, the type $\forall(\alpha)\ t \rightarrow \alpha$ would not be correct for this function, while $\forall(\alpha)\ \alpha\ t \rightarrow \alpha$ is.

6 Type inference

This extension does not add much complexity to type inference of modules: type inference is mostly unchanged except for the addition of generalization and instantiation of module types. So that type constructors do not get unified to type variables bound outside of their scope, unification under a prefix must be used. This is simple, and easy to implement (and is already implemented in Objective Caml, that allows module components to have non-closed types.) Caml modules do not always have principal signatures: some modules can be given two incompatible signatures. This does not appear to be a problem though. Indeed, if the signatures of functor parameters are considered given, principal signatures (with respect to these constraints) can be inferred. Unification under a prefix has principal solutions. So, our extension preserves this property. Furthermore, a module system with principal signatures can be extended while preserving principal signatures.

7 Imperative features

So as to ensure type safety, value-only polymorphism [9] should *a priori* be extended to module expressions. That is, only module expressions that are syntactically module values (paths and functors, but neither structures nor functor applications) should be generalized[3]. The rational is that the evaluation of these expressions cannot produce side-effects. However, this results in a important loss of expressiveness: the example of polymorphic sets in section 5 is not valid anymore. This does not appear to be a problem for classes though, as classes are usually functors.

A way to circumvent this problem is to enrich module types, so as to distinguish functors that evaluates without producing any side-effects from other functors. Indeed, it is sound to generalize the type of the result of an application of such a functor.

8 Limitations and future work

Even though our proposal combines classes and modules in one construction, the choice between the two programming styles remains: one can either provide an abstract type and functions manipulating values of this type:

```
module stack : sig type α t; ... end = struct
   type α t = α list;
   val empty = [];
   val push l x = x :: l;
   val pop (x :: l) = (x, l);
   ...
end
```

or define some methods providing the same functionalities:

```
module stack = [α] struct
   val impl = ([] : α list);
   method push =
      ς(y) fun x → {⟨impl = x :: impl⟩};
   method pop =
      ς(y) let (x :: l) = impl in (x, {⟨impl = l⟩});
   ...
end
```

(The expression $\{\langle x = e\rangle\}$ evaluates to a copy of self where the value component x is bound to the result of the evaluation of expression e.)

In this case, it is actually possible for the same module to provide both styles.

```
module stack = [α'] struct
   type α t = α list;
   val empty = [];
   val push x l = x :: l;
   val pop (x :: l) = (x, l);
   ...;
   val impl = (empty : α' t);
   method push =
      ς(y) fun x → {⟨impl = push impl x⟩};
   method pop =
      ς(y) let (x, l) = pop impl in (x, {⟨impl = l⟩});
   ...
end
```

However, for each function, a corresponding method must still be written, consisting actually (at least in this case) in just stub code calling the function.

It would be interesting to be able to write any module so that it can be used for both styles of programming, without having any stub code to write. As a first step toward this goal, one can notice that it is possible to automatically generate a module providing the functionalities of the first module *stack* from the class *stack*:

```
module stack' : sig type α t; ... end = struct
   type α t = α stack;
   val empty = new stack;
   val push s x = s#push x;
   val pop s = s#pop;
   ...
end
```

Furthermore, this wrapping can easily be generalized to any class definition: a module can be associated to any class; this module exports the type of the objects of this class (in the example above, this is $\alpha\ \alpha$), a function corresponding to the object creator (*empty*) and a function for each method (*push*, *pop*, ...), that invokes the method.

However, classes written as a wrapper around an abstract datatype seems to be a common pattern, which makes the previous wrapping much less interesting. As an example, one can consider the implementation of sets as balanced

[2] Constrained type abbreviations are presented in [7]; another variant was previously used in [1].

[3] The rule for module subtyping corresponding to generalization should also be modified.

trees. It seems more natural to define balanced trees using an abstract datatype (so that they can be manipulated using pattern matching), and then eventually provide a class that hides this implementation. But it is not clear how such a class could be generated automatically, in a regular way, in the general case. A module may define several datatypes, and only one of them should be chosen (a module for sets would probably export both the type of elements and the type of sets). The first argument of some functions may not have the right type: for instance, a function adding an element to a set could take the element, rather than the set, as first argument. A function such as set difference takes two arguments of the same type. It is somewhat arbitrary to distinguished one or the other. Another problem of such a function is that it implies that an object must access the implementation of another object (which should be private.)

9 Related works

Reppy and Riecke already proposed to encode classes in modules [8]. They use Object ML, an extension of SML with simple objects and subtyping, but no classes. Their encoding requires no further extension. As a consequence, it is somewhat verbose. For instance, a creation function must be explicitly defined; this function creates a new object and explicitly fills the different members of the object. Some syntactic sugar should be provided in order to simplify the encoding, but it is not clear how this could be done. On the other hand, it should be possible to adapt our proposal to Object ML. As Object ML classes are not polymorphic, type quantification of modules would not even be necessary.

Type quantification of module expressions has also been suggested by Stefan Kahrs [4]. His proposal differs from ours in several points. We illustrate this by an example:

$$
\begin{array}{l}
\texttt{struct} \\
\quad \texttt{module}\ z_1 = \texttt{functor}\ (z' : \texttt{sig type}\ t; \ldots\ \texttt{end}) \rightarrow \\
\qquad\qquad\qquad \texttt{struct}\ \ldots\ \texttt{end}; \\
\quad \texttt{typevar}\ \alpha; \\
\quad \texttt{type}\ t_1 = \alpha; \\
\quad \texttt{module}\ z_2 = z_1\ \big(\texttt{struct type}\ t = t_1; \ldots\ \texttt{end}\big) \\
\texttt{end}
\end{array}
$$

Type variables are explicitly introduced, using a new binder `typevar`. A type definition in the scope of a type variable implicitly depends on this type variable (which can appear on the right-hand side of the type definition, as this is the case here for type t_1.) As a consequence, when used outside of the scope of the type variable, the defined type constructor takes an additional parameter (for instance, the type constructor t_1 becomes parameterized outside the scope of the structure). This proposal seems to be roughly equivalent to ours. For instance, the previous example can be adapted as follows (the type t is made implicit, while the type t_1 is explicitly parameterized):

$$
\begin{array}{l}
\texttt{struct} \\
\quad \texttt{module}\ z_1 = [\alpha]\ \texttt{functor}\ (z' : \texttt{sig}\ \ldots\ \texttt{end}) \rightarrow \\
\qquad\qquad\qquad\quad \texttt{struct}\ \ldots\ \texttt{end}; \\
\quad \texttt{type}\ \alpha\ t_1 = \alpha; \\
\quad \texttt{module}\ z_2 = z_1\ \big(\texttt{struct}\ \ldots\ \texttt{end}\big) \\
\texttt{end}
\end{array}
$$

However, it seems more difficult to support imperative features with his proposal: in signatures, type components may need some annotations. For instance, let us assume that the structure in the definition of module z_2 above contains an exception definition: `exception` E `of` t. The type t of the exception must not be polymorphic. But, as type parameters may be implicit, this cannot be locally enforced, and must somehow appear in the signature of module z_1. Indeed, in the definition of module z_2, the type t is then bound to type t_2 which expands to type α. As a consequence, this type variable should be generalized. On the other hand, with our proposal, the exception definition would be `exception` E `of` α, and one would just have to prevent the generalization of the type variable α.

10 Conclusion

We have been able to smoothly combine the class language of Objective ML with a module language. Our proposal thus brings a better comprehension of classes and modules, and how they relates. It requires a small extension to module typing: structure components are allowed to have non-closed types, and module types can be quantified (this extension would not be necessary for a class language which does not support parametric classes.) This extension is interesting in itself.

While combining classes and modules reduces the complexity of a language providing both, in our proposal modules must still be used either as usual modules or as classes. It is still not possible to use the same definition both as a module and a class.

References

[1] Bernard Berthomieu. Programming with behaviors in an ML framework, the syntax and semantics of LCS. Research Report 93-133, LAAS-CNRS, 7, Avenue du Colonnel Roche, 31077 Toulouse, France, March 1993.

[2] Judicaël Courant. A module calculus enjoying the subject-reduction property. Research Report 96-31, LIP, October 1996.

[3] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *21st symposium Principles of Programming Languages*, pages 123–137. ACM Press, 1994.

[4] Stefan Kahrs. First-class polymorphism for ML. Technical report, University of Edinburgh, 1993.

[5] Xavier Leroy. The Objective Caml system. Software and documentation available on the Web, `http://pauillac.inria.fr/ocaml/`, 1996.

[6] Xavier Leroy. A syntactic theory of type generativity and sharing. *Journal of Functional Programming*, 6(5):667–698, 1996.

[7] Didier Rémy and Jérôme Vouillon. Objective ML: An effective object-oriented extension to ML. *Theoretical And Practice of Objects Systems*, To appear, 1998. Available at `ftp://ftp.inria.fr/INRIA/Projects/cristal/Didier.Remy/objective-ml!tapos98.ps.gz`.

[8] John H. Reppy and Jon G. Riecke. Classes in Object ML. Presented at the FOOL'3 workshop, July 1996.

[9] Andrew K. Wright. Polymorphism for imperative languages without imperative types. Technical report TR93-200, Rice University, 1993.

A Equivalence and subtyping relations

Type access path

$$\frac{(\texttt{module } z_i : \Phi) \in A}{A \vdash z_i : \Phi}$$

$$\frac{A \vdash p : (\texttt{sig } (\tau)\ \chi\ \texttt{end}) \qquad (\texttt{module } z_i : \Phi) \in \chi \qquad A + \chi \vdash \Phi \approx \Phi'}{A \vdash p.z : \Phi'}$$

$$\frac{A \vdash p : \forall(\alpha)\ \Phi}{A \vdash p : \Phi[\tau/\alpha]}$$

Type equivalence (rules for congruence, reflexivity, symmetry, transitivity omitted)

$$\frac{(\texttt{type } t_i = \tau) \in A}{A \vdash t_i \approx \tau}$$

$$\frac{A \vdash p : \texttt{sig } (\tau)\ \chi\ \texttt{end} \qquad (\texttt{type } t_i = \tau) \in \chi}{A \vdash p.t \approx \tau}$$

Module subtyping

$$\frac{A \vdash \tau_1 \approx \tau_2}{A \vdash \tau_1 <: \tau_2}$$

$$\frac{A \vdash \Phi_2' <: \Phi_1' \qquad A;\ (\texttt{module } z_i : \Phi_2') \vdash \Phi_1 <: \Phi_2}{A \vdash \texttt{functor } (z_i : \Phi_1') \to \Phi_1 <: \texttt{functor } (z_i : \Phi_2') \to \Phi_2}$$

$$\frac{A \vdash \tau_1 \approx \tau_2 \qquad \forall d_2 \in \chi_2 \cdot \exists d_1 \in \chi_1 \cdot A + \chi_1 \vdash d_1 <: d_2}{A \vdash (\texttt{sig } (\tau_1)\ \chi_1\ \texttt{end}) <: (\texttt{sig } (\tau_2)\ \chi_2\ \texttt{end})}$$

$$\frac{A \vdash T_1 <: T_2}{A \vdash (\texttt{val } x_i : T_1) <: (\texttt{val } x_i : T_2)}$$

$$\frac{A \vdash \Phi_1 <: \Phi_2}{A \vdash (\texttt{module } z_i : \Phi_1) <: (\texttt{module } z_i : \Phi_2)}$$

$$\frac{}{A \vdash (\texttt{method } y) <: (\texttt{method } y)}$$

$$\frac{}{A \vdash (\texttt{type } t_i) <: (\texttt{type } t_i)} \qquad \frac{}{A \vdash (\texttt{type } t_i = \tau) <: (\texttt{type } t_i)}$$

$$\frac{A \vdash \tau \approx \tau'}{A \vdash (\texttt{type } t_i = \tau) <: (\texttt{type } t_i = \tau')}$$

$$\frac{A \vdash t_i \approx \tau}{A \vdash (\texttt{type } t_i) <: (\texttt{type } t_i = \tau)}$$

$$\frac{A \vdash \Phi_1[\tau/\alpha] <: \Phi_2}{A \vdash \forall(\alpha)\ \Phi_1 <: \Phi_2} \qquad \frac{A \vdash \Phi_1 <: \Phi_2 \qquad \alpha \notin \mathrm{FV}(A)}{A \vdash \Phi_1 <: \forall(\alpha)\ \Phi_2}$$

$$\frac{A \vdash T_1[\tau/\alpha] <: T_2}{A \vdash \forall(\alpha)\ T_1 <: T_2} \qquad \frac{A \vdash T_1 <: T_2 \qquad \alpha \notin \mathrm{FV}(A)}{A \vdash T_1 <: \forall(\alpha)\ T_2}$$

Module type scheme equivalence

$$\frac{A \vdash \Phi_1 <: \Phi_2 \qquad A \vdash \Phi_2 <: \Phi_1}{A \vdash \Phi_1 \approx \Phi_2}$$

B Type translation

Core type (structural rules omitted)

(Type-Name)
$$\frac{(\texttt{type } t_i [= \tau\ ]) \in A}{A \vdash t : t_i}$$

(Type-Projection)
$$\frac{A \vdash \tilde{p} : \texttt{sig } (\tau)\ \chi\ \texttt{end} \qquad (\texttt{type } t_i = \tau) \in \chi}{A \vdash \tilde{p}.t : \tau}$$

(Type-Var)
$$\frac{A(\alpha) = (\texttt{var } \alpha_i = \tau)}{A \vdash \alpha : \tau}$$

(Type-Quant)
$$\frac{A;\ (\texttt{var } \alpha_i = \alpha') \vdash \tilde{T} : T \qquad \alpha' \notin \mathrm{FV}(A)}{A \vdash \forall(\alpha)\ \tilde{T} : \forall(\alpha')\ T}$$

Signature

(Sig-Empty)
$$\frac{}{A \vdash \epsilon : \epsilon}$$

(Sig-Then)
$$\frac{A \vdash \tilde{\psi} : \psi \qquad A;\ \psi \vdash \tilde{\chi} : \chi}{A \vdash (\tilde{\psi};\ \tilde{\chi}) : (\psi;\ \chi)}$$

(Sig-Value)
$$\frac{A \vdash \tilde{T} : T}{A \vdash (\texttt{val } x : \tilde{T}) : (\texttt{val } x_i = T)}$$

(Sig-Module)
$$\frac{A \vdash \tilde{\Phi} : \Phi}{A \vdash (\texttt{module } z : \tilde{\Phi}) : (\texttt{module } z_i : \Phi)}$$

(Sig-Manifest-Type)
$$\frac{A \vdash \tilde{\tau} : \tau \qquad \mathrm{FV}(\tau) = \emptyset}{A \vdash (\texttt{type } t = \tilde{\tau}) : (\texttt{type } t_i = \tau)}$$

(Sig-Abstract-Type)
$$\frac{}{A \vdash (\texttt{type } t) : (\texttt{type } t_i)}$$

(Sig-Method)
$$\frac{}{A \vdash (\texttt{method } y) : (\texttt{method } y)}$$

(Signature)
$$\frac{A \vdash \tau : \tau' \qquad A \vdash \chi : \chi'}{A \vdash (\texttt{sig } (\tau)\ \chi\ \texttt{end}) : (\texttt{sig } (\tau')\ \chi'\ \texttt{end})}$$

(Sig-Functor)
$$\frac{A \vdash \tilde{\Phi}' : \Phi' \qquad A;\ (\texttt{module } z_i : \Phi') \vdash \tilde{\Phi} : \Phi}{A \vdash (\texttt{functor } (z : \tilde{\Phi}') \to \tilde{\Phi}) : (\texttt{functor } (z : \Phi') \to \Phi)}$$

(Sig-Quant)
$$\frac{A;\ (\texttt{var } \alpha_i = \alpha') \vdash \tilde{\Phi} : \Phi \qquad \alpha' \notin \mathrm{FV}(A)}{A \vdash \forall(\alpha)\ \tilde{\Phi} : \forall(\alpha')\ \Phi}$$

C Type well-formedness

Type well-formedness ($A \vdash T$ type)

(structural rules omitted)

$$\frac{\left(\texttt{type}\ t_i[\ = \tau\ ]\right) \in A}{A \vdash t_i \text{ type}}$$

$$\frac{A \vdash p : \texttt{sig}\ (\tau)\ \chi\ \texttt{end} \qquad \left(\texttt{type}\ t_i = \tau\right) \in \chi}{A \vdash p.t \text{ type}}$$

Module type well-formedness ($A \vdash \Phi$ module type)

$$\frac{}{A \vdash \epsilon \text{ decl}} \qquad \frac{A \vdash T \text{ type} \qquad A \vdash \chi \text{ decl}}{A \vdash \left(\texttt{val}\ x_i = T;\ \chi\right) \text{ decl}}$$

$$\frac{A \vdash \chi \text{ decl}}{A \vdash \left(\texttt{method}\ y;\ \chi\right) \text{ decl}}$$

$$\frac{A \vdash \tau \text{ type} \qquad A;\ \left(\texttt{type}\ t_i = \tau\right) \vdash \chi \text{ decl}}{A \vdash \left(\texttt{type}\ t_i = \tau;\ \chi\right) \text{ decl}}$$

$$\frac{A;\ \left(\texttt{type}\ t_i\right) \vdash \chi \text{ decl}}{A \vdash \left(\texttt{type}\ t_i;\ \chi\right) \text{ decl}}$$

$$\frac{A \vdash \phi \text{ module type} \qquad A;\ \left(\texttt{module}\ z_i : \phi\right) \vdash \chi \text{ decl}}{A \vdash \left(\texttt{module}\ z_i : \phi;\ \chi\right) \text{ decl}}$$

$$\frac{\begin{array}{c} A \vdash \tau \text{ type} \qquad A \vdash \chi \text{ decl} \\ \text{no name bound twice in } \chi \end{array}}{A \vdash \left(\texttt{sig}\ (\tau)\ \chi\ \texttt{end}\right) \text{ module type}}$$

$$\frac{\begin{array}{c} A \vdash \Phi' \text{ module type} \\ A;\ \left(\texttt{module}\ z_i : \Phi'\right) \vdash \Phi \text{ module type} \end{array}}{A \vdash \left(\texttt{functor}\ (z_i : \Phi') \to \Phi\right) \text{ module type}}$$

$$\frac{A \vdash \Phi \text{ module type} \qquad \alpha \notin \mathrm{FV}(A)}{A \vdash \forall(\alpha)\ \Phi \text{ module type}}$$

D Strengthening

(structural rules omitted)

$$\begin{array}{rcl}
(\texttt{functor}\ (z_i : \Phi') \to \Phi)/p & \longrightarrow & \texttt{functor}\ (z_i : \Phi') \to \Phi/(p\ z_i) \\
(\texttt{sig}\ (\tau)\ \chi\ \texttt{end})/p & \longrightarrow & \texttt{sig}\ (\tau)\ \chi/p\ \texttt{end} \\
(\texttt{module}\ z_i : \Phi)/p & \longrightarrow & \texttt{module}\ z_i : \Phi/p.z \\
(\texttt{type}\ t_i)/p & \longrightarrow & \texttt{type}\ t_i = p.t \\
(\texttt{type}\ t_i = \tau')/p & \longrightarrow & \texttt{type}\ t_i = p.t
\end{array}$$

Virtual Types *are* Statically Safe

Mads Torgersen
Computer Science Department, Århus University
Ny Munkegade, DK-8000 Århus C, Denmark
(+45) 89 42 32 57
madst@daimi.aau.dk

Abstract

Virtual types are a combined genericity and covariance mechanism first introduced in BETA. Like most other covariant language constructs, virtual types in their original form depend on dynamic checking for type safety. This paper presents full statical type checking for virtual types, while not relying on any other special language mechanisms for safety. Good expressiveness is retained, as demonstrated by a simple but effective solution to the infamous ColourPoint problem.

1 Introduction

Madsen [11] states that "it is well known that at most two of the following three properties can be obtained at the same time:

1. Static typing.
2. Subtype substitutability.
3. Covariance."

It is the intention of this paper to demonstrate that this need not be so.

Virtual types, originally introduced in BETA[14], are a very powerful and expressive mechanism, which gives rise to covariance. In their original form, virtual types gave rise to a certain amount of dynamic typechecking, and this was adopted by a recent proposal for introducing virtual types in Java [9] by Thorup [18].

In this paper we introduce fully statical type checking of virtual types, while not compromising subtype substitutability in any way. The key to retaining expressiveness under these conditions is the ability – also present in BETA – of virtual types to be *final bound*.

2 What are virtual types

Virtual types are best thought of as symbolic names for classes. These names (or aliases) are themselves attributes of classes, and their *binding* – the class they are a name for – may be overwritten in subclasses. This overwriting – called *further binding* – must be *covariant*, i.e. to a subclass of the previous binding.

A special kind of further binding called a *final binding* has the additional property of fixing the binding of the virtual type: It may not be further bound in subclasses. Thus, in this way the potential further covariance of a virtual type may be explicitly prohibited.

2.1 An example

The "Cow Example", originating from a brilliant treatment of covariance by David Shang [17], illustrates the mechanism of virtual types and final bindings as well as serving as a basis for demonstrating their typing implications.

```
Animal = {
   FoodType <= Food;
   eat(f: FoodType) { ... }
}

Herbivore = Animal {
   FoodType <= Plants;
}

Cow = Herbivore {
   FoodType = Grass;
}
```

Figure 1: The cow example

FoodType is a virtual type. In the declaration of Animal it is *declared*, and bound to Food. This means that "an Animal has a FoodType, which is *some* subclass of Food". The consequence for the eat method also declared in Animal is that "an Animal can eat *some* kind of Food".

In the declaration of Herbivore as a subclass of Animal, FoodType is *further bound* to Plants. This means that "the FoodType of an Herbivore is *some* subclass of Plants". As a consequence "an Herbivore can eat *some* kind of Plants".

In the declaration of Cow as a subclass of Herbivore, FoodType is *final bound* to Grass. This means that "the FoodType of a Cow is *precisely* Grass". As a consequence "a Cow can eat *any* kind of Grass".

Leaving out an implementation of the eat method should emphasize the fact (already suggested by their name) that virtual types are primarily a matter of *typing* rather than *functionality*. The only functionality directly associated with virtual types is the possibility of instantiating them with the new operation.

In order to see the typing implications of virtual types, let us investigate some uses of the Animal hierarchy of Figure 1.

```
Farmer = {
  FeedCow(c: Cow, g: Grass) {
    c.eat(g);
  }
}
```

Figure 2: Cow feeding: A sure success

A Farmer, as defined if Figure 2, is able to take a cow and some grass, and give the grass to the cow. This, according to the interpretation of the Cow class given above, is quite alright, since any Cow will eat any kind of Grass.

```
BadFarmer = {
  FeedCow(c: Cow, m: Meat) {
    c.eat(m);
  }
}
```

Figure 3: Cow choking: A sure failure

At the other end, since Meat is definately not Grass, trying as the BadFarmer of Figure 3 to give it to a cow can never succeed, and is therefore a certain static type error.

Trying to generalize this mixed experience we move on to define a Keeper (Figure 4) who is able to take an animal and some food, and give the food to the animal.

```
Keeper = {
  FeedAnimal(a: Animal, f:Food) {
    a.eat(f);
  }
}
```

Figure 4: A keeper having a problem

Trying this out with a cow and some grass should work just as fine as for the Farmer above. Unfortunately, giving the cow meat or giving the grass to a lion seems to be equally possible. Thus, the FeedAnimal method of the Keeper class is type safe on some inputs but not on others. The problem is that the method assumes that *any* Animal can eat *any* kind of Food, while according to the declaration of Animal it is only guaranteed to eat *some* kind of Food.

BETA handles this optimistically by allowing the method to be statically type checked, but inserting code for performing a *dynamic* type check every time the method is invoked to ensure that only sensible (i.e. typesound) executions of a.eat(f) are allowed.

Essentially the *only* difference between the BETA approach and the one suggested in this paper is, that we do not allow such optimism. In order to buy the full static type safety that BETA doesn't have, the possibilty of writing code such as the Keeper is sacrificed. Where BETA inserts runtime type checks, we flag a static type error instead.

3 Other covariance mechanisms

This section investigates the relationship of virtual types with other language constructs giving rise to covariance. We look at the ability of virtual types to express these constructs, as well as the possibility of adding equivalents of final bindings to them, in order to try to gain the same static safety for these mechanisms as we just achieved with virtual types.

3.1 Signature covariance

In Eiffel [15], overwriting a method in a subclass allows for specialization of the method parameters, a facility one could call *signature covariance*. Since any parameter types may be specialized, only a global analysis of the program can determine whether unsafe situations arise.

It should be clear that this mechanism can be simulated with method parameters of virtual type, like f:FoodType in the eat method from Figure 1. Furthermore, more static typechecking is possible, exploiting that parameters with non-virtual or closed virtual type are guaranteed *not* to be covariant.

It is quite straightforward to imagine making languages like Eiffel safer by adding an anonymous parallel of final bindings; some kind of marker explicitly flagging a given parameter type as "not covariant" in subclasses.

3.2 Self recursive classes

[3] gives a good overview of various approaches to the problem of so-called *binary methods*; methods that are supposed to take arguments of the type of their enclosing object. One way to express this is to introduce a special keyword MyType [2] or SAME[16] which is covariant in the sense that it always denotes the class of the *actual* enclosing object - not the one where the MyType appears.

One might view (and implement) MyType as a virtual type which is automatically bound in all classes [18]. Such a solution makes static typing problematic, however, because MyType obviously cannot be final bound (except of course in a language that has final classes, such as Java). BETA lets the programmer manually specify such virtuals whenever needed [13], and final bind them at suitable places. An example of this approach is the CompareType of the ColourPoint example in Figure 12 and 13.

4 A pseudo-language

Until now virtual types have most often been described in the context of full-fledged commercial programming languages, specifically BETA and Java [14, 18]. This has the disadvantage that the typing aspects of virtual types are mangled with those of other constructs of these languages; e.g. part objects and block structure in BETA, or abstract and final classes, interfaces and constructors in Java.

For this reason we have chosen here to use an absolutely minimal pseudo-language, which contains no such features and is designed so as to let the properties of virtual types stand out for themselves.

PROGRAM	::=	CLASSDECL*
CLASSDECL	::=	*ClassId* = *ClassId* { VIRTUALDECL* VARDECL* METHODDECL* }
VIRTUALDECL	::=	*VirtualId* <= *ClassId* ;
	|	*VirtualId* = PRETYPE ;
VARDECL	::=	*VarId* : PRETYPE ;
METHODDECL	::=	*MethodId* (PARDECL*) : PRETYPE { STATEMENT* }
PARDECL	::=	*ParId* : PRETYPE
PRETYPE	::=	*ClassId*
	|	*VirtualId*
STATEMENT	::=	EXPRESSION ;
	|	EXPRESSION.*VarId* := EXPRESSION ;
	|	return EXPRESSION ;
EXPRESSION	::=	this
	|	*ParId*
	|	EXPRESSION.*VarId*
	|	EXPRESSION.*MethodId* (EXPRESSION*)
	|	new TYPE
TYPE	::=	*ClassId*
	|	EXPRESSION.*VirtualId*

Figure 5: The pseudo-language

This approach carries with it the danger that its results cannot survive subsequent exposure to a more complex environment (indeed this is acknowledged as one of the motivating factors for applying formal type theory to Java in [6]). In order to try out the type checking scheme and programming techniques suggested in this paper in a somewhat more realistic setting, statically safe virtual types are currently being added to a compiler for a subset of Java; see Section 9.

The language, sketched in figure 5, is intended to be the essential core of a statically typed, statically scoped, class-based, single-inheritance object-oriented language. The only thing special is the presence of virtual types.

For clarity, and to ease the formulation of type rules, the syntax is somewhat restrictive. Several kinds of programming constructs and syntactic sugar with little impact on type checking might easily be added, and indeed many of the program examples throughout this paper belong to a more permissive superset. In the course of the following description some obvious relaxations and additions will be noted.

4.1 Declarations

A *program* is a number of class declarations. Each *class* has one superclass, which could have been optional. (Multiple inheritance would seriously complicate things and is not considered.) It furthermore has a body of attribute declarations, of which there are three kinds: Virtual types, instance variables and methods.

Virtual types are declared and further bound with "<=" and final bound with "=". A final binding prohibits further binding in subclasses. Virtual types are covariant, which means that a binding must always specialize the previous one. A virtual type V can be final bound to another one V', meaning that V will automatically have the same binding as V' in all subclasses. A virtual type is *closed* in a class if it is final bound to a class or another closed virtual type. Otherwise it is *open*.

Instance variables are mutable object references. Their declaration may not be overridden in subclasses. Immutable references could be added, but would complicate things due to the need for an initialization mechanism. As will be evident below, immutable references play a special role in the type checking of expressions, but method parameters, which *are* immutable, will suffice to demonstrate this point.

Methods are declared with a number of formal parameters, a return type (which in a "real" language would have been optional) and a body of statements. The body may be overridden in subclasses, but the signature (formal parameters and return type) cannot be changed. As already mentioned, formal parameters are immutable object references. Because covariance on signatures is not allowed, statically resolved method overloading as in $C++$ [7] could be added without problems. For the same reason a super construct for calling overwritten methods as in Smalltalk and Java [8, 9] would have identical type properties to this.

4.2 Statements and expressions

The syntax in figure 5 contains only statements and expressions that are important from a typing point of view. This means that there are no control structures, literal expressions or binary operators. Likewise, local variables within statements are omitted.

Only method parameters and the special constant this, denoting the current instance of the enclosing class, may be accessed directly, while all kinds of object attributes – variables, methods and virtual types – must be accessed via a *dot expression* (as in $e.a$, $e.m()$ and new $e.V$ respectively). Of course a free *VarId* a could easily be treated as syntactic sugar for this.a, as is normal practice.

4.3 Runtime behaviour

At runtime, the evaluation of an expression results in an object. Every object has a class, and is created with the primitive `new`. If the argument to `new` is a simple class, the new object gets that class. If is is of the form `e.V`, where `V` is a virtual type, then `e` evaluates to some runtime object, `o`. The binding of `V` in the class of `o` will then be the class of the new object.

The implementation of object creation requires the binding of virtual types to be represented at runtime. If the language included casting or typecase constructs, they too would need such runtime information, as would of course implicit runtime type checks as used in BETA.

5 Static type checking with virtual types

The purpose of this section is to give a detailed constructive description of how the type checking mechanism is designed to achieve static safety.

To motivate the design we first review what it takes to be type safe in terms of program *execution* and then move on to discuss in an abstract way how this can be checked statically in terms of program *inspection*.

On the basis of this, a detailed account of the proposed type checking mechanism is given. Type rules are presented only for statements and expressions, while the typing constraints on declarations are described more informally. Apart from a complete set of type rules, an actual proof of type safety would also require the specification of a formal semantics of the language.

5.1 Type safety

We assume the following definitions of type safety:

- A program execution is type safe if it no attempt is made to access an attribute of an object that the object doesn't have.
- A program is type safe if all executions of it are type safe.
- A language is type safe if any program in the language is type safe.

Additionally, type safety is *static*, if it does not depend on checks performed during program execution.

As for the runtime behaviour of a language implementation it seems fair to assume that

- The class of an object does not change.
- An object actually has the attributes declared in its class.

Thus it suffices to verify that objects are used according to their class. This in turn follows if each reference (instance variable or method parameter) has an associated *qualification* so that

- The qualification of a reference is a superclass of the class of the object it refers to.
- Only attributes declared in the qualification of a given reference are accessed on the object it refers to.

This rendering is operational in the sense that it tells us what to check for:

- When assigning an object to a reference (by variable assignment or method parameter passing), check for the appropriate qualification.
- When accessing an attribute of a reference, check the class for an appropriate declaration.

We are now ready to move to static typing. Inspecting a given statement in the program, we can establish its static safeness (the fact that it preserves static type safety) if we can conservatively assert that the above will hold *whenever* it is executed.

A natural way to do this is to let expressions and references have *static types* which summarize the possible classes they may have at runtime. This will allow us to statically check type safety in a way that nicely parallels the above:

- Assignment is checked with the use of a *subtype* relation between *static* types, which asserts that the needed subclass relation will always hold at runtime.
- Attribute access is checked using the most specific statically known common superclass of all classes denoted by the type.

If a static type is *covariant*, it generally doesn't have any guaranteed subtypes at compile time (the only exceptions being quite specific special cases, as exemplified in Section 5.3 and Figure 6). This is the situation where BETA would introduce runtime checks.

The pivotal point of the following is the fleshing out of the static type concept and subtype relation for the pseudo-language.

5.2 Basic assumptions

In the following, different names denote different kinds of entities, according to this table:

name	*is used for*
C_0, C, C'	classes
A	attributes
V	virtual types
T, T_i	pretypes
M_0, M	methods
S	statements
$a,$	instance variables
p	parameters
e, e_i	expressions
τ, τ', τ_i	types
i, n	numbers

Attributes is a common designation for virtual types, instance variables and methods. Types and pretypes will be further explained below.

5.3 Pretypes and types

With all that in place, we can now attack the central problem: What should a static type be, in order to facilitate full static type checking, yet be as permissive as possible, so as not to inhibit expressiveness.

A good starting point is to see what types we might assign to a `new` expression, since this is where objects begin their existence. To go with "`new` C" it seems obvious to introduce a plain class type, denoted by $\langle C \rangle$.

What should be the type of "`new` $e.V$"? Since the binding of V depends recursively on the type of e (say τ), the only way not to throw away potentially useful type information is to keep τ along with V in the type of the expression. Such a type is written $\langle\tau, V\rangle$.

This scheme matches well with the fact that the so-called *pretype* of e.g. variable declarations (see Figure 5) can be either a class or a virtual type. When a variable with class pretype (say $a : C$) is used within a method body (as in "this.a"), its type should of course be $\langle C\rangle$.

But what should be the type of a variable with virtual pretype (say $a : V$) within a method body? The type is not completely determined by the pretype, but depends on the type of the object that a resides in – its *location*. This incompleteness is why the type part of a reference declaration is called a *pre*-type.

The type scheme $\{\langle C\rangle, \langle\tau, V\rangle\}$ generally works well, but there is one nuisance still left to cope with. In general, we have no option but to forbid assignment to references of open virtual type. Reverting to the animal example for a moment, however, take a look at the class declaration in Figure 6: Given an animal, the keeper feeds it its own kind of food! There would be nothing wrong with accepting this, since the parameter type of eat – although definately virtual – is obviously the same as the one produced by new a.FoodType.

```
Keeper = {
  Feed(a: Animal) {
    a.eat(new a.FoodType);
  }
}
```

Figure 6: A keeper at a zoo

The reason is that we are refering to the same virtual type in the same object – the latter "same" guaranteed by the immutability of method parameters. To allow for this special case, we introduce an extra kind of type $\langle c\rangle$ for constant references c; in our case parameters or this. Their type is simply themselves.

5.4 Environments

The *program environment* $\mathcal{P}$ contains the class declarations of the program, and the subclass relation between them. It defines the following relations:

relation	*means*
$C \in \mathcal{P}$	C is declared in $\mathcal{P}$
$\mathcal{P} \vdash C \leftarrow C'$	C is a subclass of C'

A *class environment* contains the attributes of the class. It defines the following relations:

relation	*means*
$\mathcal{P}, C_0 \vdash V \texttt{<=} C$	V is open, bound to C
$\mathcal{P}, C_0 \vdash V \texttt{=} T$	V is final bound to T
$\mathcal{P}, C_0 \vdash a : T$	a is declared to T
$\mathcal{P}, C_0 \vdash M : T_1 \times ... \times T_n \to T$	signature of M

A *method environment* contains the parameters and body of the method. It defines the following relations:

relation	*means*
$\mathcal{P}, C_0, M_0 \vdash e : \tau$	e has type τ
$\mathcal{P}, C_0, M_0 \vdash \tau <: \tau'$	τ is a subtype of τ'
$\mathcal{P}, C_0, M_0 \vdash S$	S typechecks

5.5 Checking declarations

Type rules for the declaration of classes and attributes have been left out. Thus a description lacks of how the various environments are constructed and their well-formedness maintained (e.g. according to Section 4.1). While this is certainly nontrivial, it is generally not profoundly influenced by the presence of virtual types. One exception to this is final binding of one virtual type to another. This must be non-circular in order for the *Type* function of Figure 7 to terminate.

5.6 Type inference rules

Finally we can give precise meaning to the type checking requirements treated throughout this section.

As explained above (Section 5.3), the type of a reference in the context of a given method body must be computed from its pretype and the type of its location. This computation is handled by the *Type* function, which is defined by means of inference rules in Figure 7. Notice that the computation for a finally bound virtual pretype is simply propagated to its binding - regardless of whether this binding is itself virtual.

$$\mathit{Type}_{class}\quad \frac{\vdash C \in \mathcal{P}}{\mathcal{P}, C_0, M_0 \vdash \mathit{Type}(C, \tau) = \langle C\rangle}$$

$$\mathit{Type}_{virtual}\quad \frac{\mathcal{P}, C_0, M_0 \vdash C' = \mathit{Class}(\tau) \qquad \mathcal{P}, C' \vdash V \texttt{<=} C}{\mathcal{P}, C_0, M_0 \vdash \mathit{Type}(V, \tau) = \langle\tau, V\rangle}$$

$$\mathit{Type}_{final}\quad \frac{\mathcal{P}, C_0, M_0 \vdash C' = \mathit{Class}(\tau) \qquad \mathcal{P}, C' \vdash V \texttt{=} T}{\mathcal{P}, C_0, M_0 \vdash \mathit{Type}(V, \tau) = \mathit{Type}(T, \tau)}$$

Figure 7: The *Type* function

As concluded in Section 5.1, in order to type check attribute access of an expression e, we need to find the most specific superclass of the type of e. This is done by the *Class* function, which is defined in Figure 8.

$$\mathit{Class}_{class}\quad \frac{\vdash C \in \mathcal{P}}{\mathcal{P}, C_0, M_0 \vdash \mathit{Class}\langle C\rangle = C}$$

$$\mathit{Class}_{virtual}\quad \frac{\mathcal{P}, C_0, M_0 \vdash C' = \mathit{Class}(\tau) \qquad \mathcal{P}, C' \vdash V \texttt{<=} C}{\mathcal{P}, C_0, M_0 \vdash \mathit{Class}\langle\tau, V\rangle = C}$$

$$\mathit{Class}_{final}\quad \frac{\mathcal{P}, C_0, M_0 \vdash C' = \mathit{Class}(\tau) \qquad \mathcal{P}, C' \vdash V \texttt{=} T \qquad \mathcal{P}, C_0, M_0 \vdash C = \mathit{Class}(\mathit{Type}(T, \tau))}{\mathcal{P}, C_0, M_0 \vdash \mathit{Class}\langle\tau, V\rangle = C}$$

$$\mathit{Class}_{this}\quad \mathcal{P}, C_0, M_0 \vdash \mathit{Class}\langle \mathsf{this}\rangle = C_0$$

$$\mathit{Class}_{par}\quad \frac{\mathcal{P}, C_0 \vdash (p : T) \in M_0 \qquad \mathcal{P}, C_0, M_0 \vdash C = \mathit{Class}(\mathit{Type}(T, \langle C_0\rangle))}{\mathcal{P}, C_0, M_0 \vdash \mathit{Class}\langle p\rangle = C}$$

Figure 8: The *Class* function

The subtyping relation $<:$ is defined in Figure 9. It generally allows only assignments to references with class type, and even that of course only if the type of the assigned value fits. Notice however, the special treatment of virtual types within constant references, which was motivated by the Keeper example above (Figure 6).

$$\mathit{Subtype}_{class}\quad \frac{\mathcal{P}, C_0, M_0 \vdash C' = \mathit{Class}(\tau) \qquad \mathcal{P} \vdash C' \leftarrow C}{\mathcal{P}, C_0, M_0 \vdash \tau <: \langle C \rangle}$$

$$\mathit{Subtype}_{this.V}\quad \mathcal{P}, C_0, M_0 \vdash \langle \mathsf{this}, V \rangle <: \langle \mathsf{this}, V \rangle$$

$$\mathit{Subtype}_{par.V}\quad \frac{\mathcal{P}, C_0 \vdash p \in M_0}{\mathcal{P}, C_0, M_0 \vdash \langle p, V \rangle <: \langle p, V \rangle}$$

Figure 9: The subtype relation <:

The type rules proper (Figures 10 and 11) are pretty much standard, since most of the virtual type specific "magic" appears in the above helper functions and the subtype relation.

$$\mathit{Exp}_{this}\quad \mathcal{P}, C_0, M_0 \vdash \mathsf{this} : \langle \mathsf{this} \rangle$$

$$\mathit{Exp}_{par}\quad \frac{\mathcal{P}, C_0 \vdash p \in M_0}{\mathcal{P}, C_0, M_0 \vdash p : \langle p \rangle}$$

$$\mathit{Exp}_{var}\quad \frac{\mathcal{P}, C_0, M_0 \vdash e : \tau' \qquad \mathcal{P}, C_0, M_0 \vdash C = \mathit{Class}(\tau') \qquad \mathcal{P}, C \vdash a : T \qquad \mathcal{P}, C_0, M_0 \vdash \tau = \mathit{Type}(T, \tau')}{\mathcal{P}, C_0, M_0 \vdash e.a : \tau}$$

$$\mathit{Exp}_{app}\quad \frac{\begin{array}{l}\mathcal{P}, C_0, M_0 \vdash e : \tau' \\ \mathcal{P}, C_0, M_0 \vdash C = \mathit{Class}(\tau') \\ \mathcal{P}, C \vdash M : T_1 \times \ldots \times T_n \rightarrow T \\ \mathcal{P}, C_0, M_0 \vdash \tau = \mathit{Type}(T, \tau') \\ \mathcal{P}, C_0, M_0 \vdash e_i : \tau_i \quad 1 \leq i \leq n \\ \mathcal{P}, C_0, M_0 \vdash \tau_i <: \mathit{Type}(T_i, \tau') \quad 1 \leq i \leq n\end{array}}{\mathcal{P}, C_0, M_0 \vdash e.M(e_1, \ldots, e_n) : \tau}$$

$$\mathit{Exp}_{new_C}\quad \frac{\vdash C \in \mathcal{P}}{\mathcal{P}, C_0, M_0 \vdash \mathsf{new}\ C : \langle C \rangle}$$

$$\mathit{Exp}_{new_V}\quad \frac{\mathcal{P}, C_0, M_0 \vdash e : \tau' \qquad \mathcal{P}, C_0, M_0 \vdash \tau = \mathit{Type}(V, \tau')}{\mathcal{P}, C_0, M_0 \vdash \mathsf{new}\ e.V : \tau}$$

Figure 10: Expression types

6 Expressiveness

To demonstrate that full static typing of virtual types does not sacrifice too much expressiveness, we mainly focus on the covariance problem as exemplified by the now classical ColourPoint example.

6.1 Points and ColourPoints

A standard example in the literature on covariance (see e.g. [11, 3, 5]) is the so called ColourPoint problem, which is shown in Figure 12. It is usually stated using signature covariance or self recursive types (see Section 3), but is here refrased with virtual types.

The idea is that both classes are concrete[1]. The reason for ColourPoint to inherit Point supposedly is twofold: To reuse its implementation, and to make references to Point able to polymorphically also refer to ColourPoints.

Covariance comes into play with the method equal, which expects its argument to be of the virtual type CompareType. In ColourPoint, CompareType has to be further (or in this case final) bound to ColourPoint so that the equal method can take its specific attributes into consideration.

Of course, with this structure, calling the equal method on a reference to a Point is generally not statically type safe precisely because the reference might contain instances of subclasses (indeed of ColourPoint), which further bind CompareType uncontrollably. Alas, we cannot safely compare two Points for equality!

This situation is usually regarded as a shortage of the covariance mechanism. we prefer to view it as a modelling mistake: If a Point is something that can be compared with another Point for equality, then a ColourPoint is not a subtype of Point – since it cannot. Hence, it should not be a subclass either.

[1] An *abstract* class is a class that is not supposed to be instantiated, but only functions as a superclass. By contrast, a *concrete* class is one that may have instances.

$$\mathit{Stm}_{exp}\quad \frac{\mathcal{P}, C_0, M_0 \vdash e : \tau}{\mathcal{P}, C_0, M_0 \vdash e;}$$

$$\mathit{Stm}_{assign}\quad \frac{\begin{array}{l}\mathcal{P}, C_0, M_0 \vdash e' : \tau' \\ \mathcal{P}, C_0, M_0 \vdash C = \mathit{Class}(\tau') \\ \mathcal{P}, C \vdash a : T \\ \mathcal{P}, C_0, M_0 \vdash e : \tau \\ \mathcal{P}, C_0, M_0 \vdash \tau <: \mathit{Type}(T, \tau')\end{array}}{\mathcal{P}, C_0, M_0 \vdash e'.a := e;}$$

$$\mathit{Stm}_{return}\quad \frac{\begin{array}{l}\mathcal{P}, C_0, M_0 \vdash e : \tau \\ \mathcal{P}, C_0 \vdash M_0 : T_1 \times \ldots \times T_n \rightarrow T \\ \mathcal{P}, C_0, M_0 \vdash \tau <: \mathit{Type}(T, \langle C_0 \rangle)\end{array}}{\mathcal{P}, C_0, M_0 \vdash \mathsf{return}\ e;}$$

Figure 11: Statements

```
Point = {
  x,y: Integer;
  CompareType <= Point;
  equal(p: CompareType): Bool {
    return (this.x=p.x) and (this.y=p.y);
  }
}

ColourPoint = Point {
  c: Color;
  CompareType = ColourPoint;
  equal(p: CompareType): Bool {
    return super.equal(p) and (this.c = p.c);
  }
}
```

Figure 12: The usual colour point problem

6.2 A solution

Instead, the close relationship between the two should be captured by a common abstract superclass as in Figure 13.

All desirable properties of the original approach are retained: Point and ColourPoint still share most of their implementation, References capable of polymorphically referring both can still be declared, now using AbstractPoint.

```
AbstractPoint = {
   x,y: Integer;
   CompareType <= Point;
   equal(p: CompareType): Bool {
      return (this.x=p.x) and (this.y=p.y);
   }
}

Point = AbstractPoint {
   CompareType = Point;
}

ColourPoint = AbstractPoint {
   c: Color;
   CompareType = ColourPoint;
   equal(p: CompareType): Bool {
      return super.equal(p) and (this.c = p.c);
   }
}
```

Figure 13: A slight change fixes the problem

On top of this comes the fact that Point may now be used in a safe way, and that references to Points that can *not* contain ColourPoints are now possible.

```
AbstractPoint = {
   x,y: Integer;
   CompareType <= Point;
   equal(p: CompareType): Bool {
      return (this.x=p.x) and (this.y=p.y);
   }
}

Point = AbstractPoint {
   CompareType = AbstractPoint;
}

ColourPoint = AbstractPoint {
   c: Color;
   CompareType = AbstractPoint;
   }
```

Figure 14: A different choice of semantics

Furthermore the programmer now has the full choice of whether or not to allow Points and ColourPoints to be compared with each other. If this should be possible, these classes should just final bind CompareType to AbstractPoint instead of to themselves, as shown in Figure 14. The two classes could even have different semantics, so that a Point will accept a ColourPoint, but not vice versa – the choice belongs entirely to the programmer.

This restructuring is always possible: If you need a concrete version of a class that has open virtual types, make a concrete subclass of it, where you final bind all the virtuals. If the creation of such trivial subclasses proves too tedious, one might consider simply adding a language construct for doing it automatically.

In our opinion, this cuts the Gordian knot.

7 Related work

The covariance problem as exemplified through the ColourPoint example, has been treated in numerous ways over the past decade or so. This section relates some of the proposed solutions to the one presented here.

7.1 Separating subclassing and subtyping

Due to the covariance problem, it was proposed in [4] to separate the subtype and subclass relations. This has the appeal that a variable declared by a given type does not need to be able to hold all subclasses of an associated class, but only those that are known to be substitutable for the declared type.

In Emerald [10] and many theoretical languages (such as $\mathcal{LOOM}$ [2]) this lead to a complete separation of classes (that describe implementation) and types (that describe interface), which in turn leads to quite a specification overhead when types and classes coincide.

To avoid this, Sather [16] reunifies types and classes, and only separate the two *relations*, subtyping and subclassing.

Clearly, stepping down on the demand that subclassing yields subtyping, gives new possiblities for static type checking. It is not without problems however – keeping track of two different but closely related relations can be confusing in practice.

A more profound objection seen from our point of view aims at the way this solution views inheritance: Primarily as a code reuse mechanism. Instead of modelling a conceptually clear specialization relation, subclassing is an *ad hoc* relationship between classes. This is why the need is felt for a seperate, "cleaner" relation.

From the perspective of this article, subtyping and the conceptual specialization relation coincide, and inheritance should only be allowed to be used in this way. As seen in Section 6, this does not prohibit the reuse of code, it just requires it to occur in a slightly more structured way: as inheritance of a common ancestor rather that as conceptually unsound inheritance from "cows to horses".

7.2 Monomorphic types

Covariance is only a problem because references to classes with covariant parameter types may contain instances of subtypes. Some solutions therefore introduce various constructs to avoid the unfortunate consequences of subtype substitutability in certain cases – in other words to have monomorphically as well as polymorphically typed variables.

Building on a proposal by Dodani and Tsai [5], Sather [16] divides classes into abstract and concrete, of which only the latter can be instantiated. While concrete classes can have subclasses, they do not have any subtypes. For this reason a variable qualifed by a concrete type can only refer objects of *exactly* that class.

This is reminiscent of Java's final classes [9], which prohibit further specialization, and thus make references qualified by them monomorphic.

A different way to do this is to let the references themselves be declared type exact. This is the case with part objects in Beta [12] and with exact types in $\mathcal{LOOM}$, where subtyping has been completely dropped in favour of matching; see below.

For describing instances of concrete classes, which in a natural way often are (or should be) leaf classes, monomorphism seems like a sensible idea, although it appears to be somewhat vulnarable to later redesigns of the kind where one concrete class is turned into an abstract one to introduce different variants as subclasses.

The reason why it is not considered in our langauage is that it is not necessary. Since all virtual types can be final bound, a concrete subclass without covariant parameters can aways be declared. As opposed to a final class, such a "finalized" class (like Cow in Section 2.1) can have subclasses (like DairyCow, Hereford etc.), and references qualifed by it can refer to instances of these subclasses.

7.3 Matching

Another approach is to say: well, if inheritance does not give us subtyping, what *does* it give us then? The answer is *matching* [1, 2]. In $\mathcal{LOOM}$ references can be qualifed with so called "hash types", say $\#C$, which cover any classes that *match* C, that is, all subclasses of C. A reference to a hash type is restricted so that only methods without covariant parameters can be called on it. Therefore the covariance problem does not occur.

In $\mathcal{LOOM}$, If you want to refer only to Points you use the type Point (which is monomorphic), while if you want to refer also to ColourPoints you use the type #Point, but are then not allowed to call a covariant equal method on it.

Notice how closely this corresponds to Point and AbstractPoint respectively in our solution. The only difference seems to be that we have to declare an extra class, but on the other hand get polymorphism on references to Point, plus the ability to call equal on AbstractPoint in special situations, as described in Section 5.3.

8 Genericity

This paper has mainly focused on the covariance issues of virtual types.

Virtual types as an alternative *genericity mechanism* to parameterized types (as in Eiffel [15], C++ [7] and many others) is usually criticized because of it's alleged dependence on dynamic typechecking. With this proposal that argument loses validity – as for static type safety, virtual types must now be seen as an equally valid choice.

Yet, rather than looking at virtual and parameterized types as contenders, it is compelling to ponder the possibilities of combining the strengths of the two approaches. While it is certainly outside the scope of this paper to give a proper proposal for such a fusion, it should be noted in passing, that the statical safety of virtual types brings it a lot closer.

9 Riding the Java Train

Java currently supports neither genericity nor covariance. Since these are issues that many people have opinions about, a lot of different proposals are currently being made for adding such facilities one way or the other.

And yes, a proposal for adding virtual types to Java has also come forth [18]. While blending very elegantly into the syntactic framework of Java, and – as it seems – neatly fitting the virtual types in with particular Java constructs such as interfaces and abstract classes, this otherwise convincing proposal has one serious drawback: It requires implicit dynamic type checking.

In BETA all so called reverse assignments[2] are statically allowed, but subjected to a runtime type check. The runtime checking introduced by virtual types fits nicely into this scheme, and can be seen as a special case.

Java, however, is a fully statically type checked language. With the one unfortunate exception of arrays, which are covariant and require dynamic checks, runtime checking of types can only come about as a result of *explicit* type casts in the code. The introduction of *implicit* runtime checks would profoundly violate this very deliberate principle of Java.

This is where the cavalry comes in. If the type checking scheme proposed here expands well to Java, this last problem is overcome. To gain further insight into this, statically safe virtual types are currently being added to an experimental compiler for the JOOS subset of Java (generously provided by Michael Schwartzbach and Laurie Hendren). This implementation project will serve as a proof of concept both of the feasibility of virtual types in Java and of the practicality of the type checking scheme of this paper.

10 Conclusion

The main conclusion to draw from this paper is that covariance, static type safety and subtype substitutability are not incompatible after all. But to make them coexist without losing out fatally on expressiveness, a mechanism must be provided to control covariance, and programmers need to be aware of its impact on type checking.

The type checking framework in Section 5 is to the best of our knowledge the first in-depth account of a type checking mechanism for a language with virtual types. Since the permission of dynamic checks is merely a matter of modifying the subtype relation $<:$, the approach might inspire a more formal description of e.g. the BETA type system, although admittedly the presence of block structure probably complicates matters somewhat.

While resting heavily on the presence of final bindings for safety, the proposed solution to the ColourPoint problem (Figure 13) in our oppinion also points to a more philosophical consideration. The inheritance relation prescribed by the original version reflects a simple minded desire for code reuse. Whether or not it is meaningful to say that ColourPoints actually *are* Points, and to what extend sensible semantics for comparing the two different kinds of objects may exist, is not at all considered.

Taking a *conceptual* view, we think that – even without the type safety – the new version is how we would have modelled the situation. This is just one example of what in our experience – but this *is* just a gut feeling – seems to be a common pattern: That conceptual modelling and type-theoretic soundness tend to go hand in hand.

[2] Assignments from general to more special types

References

[1] ABADI, M., AND CARDELLI, L. On subtyping and matching. In *ECOOP'95 – Object-Oriented Programming* (1995), pp. 145–167.

[2] BRUCE, K. B. Subtyping is not a good "match" for object-oriented languages. In *ECOOP'97 – Object-Oriented Programming* (1997), M. Aksit and S. Matsuoka, Eds., no. 1241 in Lecture Notes on Computer Science, Springer, pp. 104–127.

[3] BRUCE, K. B., CARDELLI, L., CASTAGNA, G., THE HOPKINS OBJECT GROUP, LEAVENS, G. T., AND PIERCE, B. On binary methods. *Theory and Practice of Object Systems*, 1 (1995), 221–242.

[4] Cook, W. R., Hill, W. L., and Canning, P. S. Inheritance is not subtyping. In *ACM Conference on Principles of Programming Languages (POPL'90)* (1990), ACM Press, pp. 125–135.

[5] Dodani, M., and Tsai, C.-S. Acts: A type system for object-oriented programming based on abstract and concrete classes. In *ECOOP'92 – Object-Oriented Programming* (1992), O. L. Madsen, Ed., no. 615 in Lecture Notes in Computer Science, Springer, pp. 309–328.

[6] Drossopoulou, S., and Eisenbach, S. Java is type safe - probably. In *ECOOP'97 – Object-Oriented Programming* (1997), M. Aksit and S. Matsuoka, Eds., no. 1241 in Lecture Notes on Computer Science, Springer, pp. 389–418.

[7] Ellis, M. A., and Stroustrup, B. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[8] Goldberg, A., and Robson, D. *Smalltalk-80 – The Language*. Addison-Wesley, 1989.

[9] Gosling, J., Joy, B., and Steele, G. *The Java Language Specification*. Addison-Wesley, 1996.

[10] Hutchinson, N. *Emerald: An Object-Oriented Language for Distributed Programming*. PhD thesis, Department of Computer Science, University of Washington, Seattle, WA, January 1987.

[11] Madsen, O. L. Open issues in object-oriented programming–a scandinavian perspective. *Software–Practice and Experience 25*, S4 (December 1995).

[12] Madsen, O. L., Magnusson, B., and Møller-Pedersen, B. Strong typing of object-oriented languages revisited. In *Proceedings of OOPSLA'90* (Ottawa, Canada, 1990), SIGPLAN, ACM Press.

[13] Madsen, O. L., and Møller-Pedersen, B. Virtual classes: A powerful mechanism in object-oriented programming. In *Proceedings of OOPSLA'89* (1989), SIGPLAN, ACM Press.

[14] Madsen, O. L., Møller-Pedersen, B., and Nygaard, K. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, 1993.

[15] Meyer, B. *Object-Oriented Software Construction*. Prentice Hall International Series in Computer Science. Prentice Hall, Englewood Cliffs, NJ, 1988.

[16] Omohundro, S. The Sather Programming Language. *Dr. Dobb's Journal 18*, 11 (October 1993).

[17] Shang, D. Are cows animals? *Object Currents 1*, 1 (January 1996). http://www.sigs.com/objectcurrents/.

[18] Thorup, K. K. Genericity in java with virtual types. In *ECOOP'97 – Object-Oriented Programming* (1997), M. Akit and S. Matsuoka, Eds., no. 1241 in Lecture Notes on Computer Science, Springer, pp. 444–471.

A statically safe alternative to virtual types
Extended Abstract

Kim B. Bruce*
Williams College
kim@cs.williams.edu

Martin Odersky
University of South Australia
odersky@cis.unisa.edu.au

Philip Wadler
Bell Labs, Lucent Technologies
wadler@research.bell-labs.com

Abstract

Parametric types and virtual types have recently been proposed as extensions to Java to support genericity. In this paper we examine both in order to investigate the strengths and weaknesses of each. We suggest a variant of virtual types which has similar expressiveness, but supports safe static type checking. This results in a language in which both parametric types and virtual types are well-integrated, and which is statically type-safe.

Keywords: Language design, virtual types, parametric polymorphism, static type checking

1 Introduction

The first step to a good answer is a good question. This note raises (and suggests an answer to) the question: Can the best features of parametric types and virtual types be integrated?

Parametric types and virtual types have both been proposed as extensions to Java, and address roughly similar issues. Parametric types have been proposed by Myers, Bank, and Liskov [MBL97], Agesen, Freund, and Mitchell [AFM97], by Bruce [Bru97], and by Odersky and Wadler as part of Pizza [OW97]. Virtual types have been proposed by Thorup [Tho97].

Parametric types and virtual types have complementary strengths. Parametric types are especially useful for collection types, such as lists or sets. (Users of C++ templates and collection classes will know many examples [KM96].) Virtual types are especially useful for families of types, such as the Subject/Observer family. (Users of design patterns will know many examples [GRJV94].)

The first step to a good question is a good example. This note presents two paradigmatic examples. The first, zip, demonstrates the strengths of parametric types. The second, lists with lengths, demonstrates the strengths of virtual types; we illustrate the important case of mutual recursion by extending this example to alternating lists. In each case, the program structure made easy by one approach can be mimicked by the other, but with difficulty.

In an attempt to bring together the strengths of each of these techniques, we present a variant of the virtual types notation that is statically type-safe. The new notation generalizes the `MyType` notation of Bruce and others to apply to mutually recursive types. This notation can be considered as a variant of the proposal for handling inheritance in the presence of mutual recursion put forward by Palsberg and Schwartzbach [PS94].

The ideas reported here gestated in the deliberations of a mailing list organized by Ole Agesen to discuss extensions of Java that support generic types. Gilad Bracha, Corky Cartwright, Guy Steele, and Kresten Thorup made particularly significant contributions.

The rest of the paper is organized as follows. Section 1 presents examples which are easy to express with parametric types, but are hard to express with virtual types. Section 2 presents an example that is easy to express with virtual types, but hard to express with parametric types. To highlight comparisons we write each example using both notations.

In Section 3 we propose a notation which is similar to virtual types, but which can be statically type checked. Section 4 shows how this can be combined with parametric types in a smooth way. Section 5 summarizes our results. Appendices provide another example of the new notation and provide more details on particular features of the language.

2 Strengths of parametric types

Here is something that is easy to do with parametric types, but hard to do with virtual types. The example is given in Pizza, with type parameters written between angle brackets <...>.

```
public class Pair<Fst, Snd> {
  public Fst fst;
  public Snd snd;
}
```


*Kim Bruce's research was partially supported by NSF grant CCR-9424123.

```
public class List<A> {
  ...
  public <B> List<Pair<A,B>> zip(List<B> yl) {...}
}
```

The types here are so expressive that it's easy to guess what zip does: pair corresponding elements of two lists. Thus if `xl` is a list of integers (say `[1,2,3]`) and `yl` is a list of strings (say `["a", "b", "c"]`), then `xl.zip(yl)` is a list of pairs of integers and strings (say `[(1,"a"),(2,"b"),(3,"c")]`).

This example is especially compact because both the class `List` and the method `zip` are parameterized (by `<A>` and `<B>` respectively). Some proposals for parametric types (such as Myer, Bank, and Liskov [MBL97] and Agesen, Freund, and Mitchell [AFM97]) allow classes but not methods to be parametric. In this case, the method `zip` can be defined via an inner class with the appropriate type parameters.

```
public class Pair<Fst, Snd> {
  public Fst fst;
  public Snd snd;
}

public class List<A> {
  ...
  public class Zipper<B> {
    public List<Pair<A,B>> zip(List<B> yl) {...}
  }
}
```

Whereas before one wrote `xl.zip(yl)`, now one would write `xl.new Zipper().zip(yl)`. (Some proposals require explicit types in expressions that create objects, so this would become `xl.new Zipper<B>().zip(yl)` when `yl` has type `List<B>`.) This is more awkward and performs an extra allocation, but permits essentially the same program structure.

It's a frustrating exercise to attempt to express the same information with virtual types. Here is an attempt to do so.

```
public class Pair {
  typedef Fst as Object;
  typedef Snd as Object;
  public Fst fst; public Snd snd;
}

public class List {
  typedef A as Object;
  public class Zipper {
    typedef B as Object;
    public class PairAB extends Pair {
      typedef Fst as A;
      typedef Snd as B;
    }
    public class ListB extends List {
      typedef Elt as B;
    }
    public class ListPairAB extends List {
      typedef Elt as PairAB;
    }
    public ListPairAB zip (ListB yl) { ... }
  }
}
```

The proliferation of class definitions is bad enough, but there is a more serious problem: each argument to `zip` must explicitly extend the class `List.Zipper.ListB`. So this solution is awkward and of restricted applicability.

2.1 Collections and subtyping

The following observation is due to Gilad Bracha.

Often one wants one collection class to inherit from another. For instance, the class for lists may inherit from a more general class for collections.

This is easy to arrange with parametric types. Here is the code outline in Pizza.

```
public class Collection<A> {...}

public class List<A> extends Collection<A> {...}
```

Now one can pass, say, an argument of type `List<String>` where one of type `Collection<String>` is expected.

Again, this is more problematic with virtual types. The obvious approach does not work.

```
public class Collection {
  typedef A as Object;
  ...
}

public class List extends Collection { ... }

public class StringCollection extends Collection {
  typedef A as String;
}

public class StringList extends List {
  typedef A as String;
}
```

The problem here is that `StringList` does not extend `StringCollection`, so the former cannot be passed where the latter is expected.

One can do a little better by exploiting inner classes.

```
public class Collection {
  typedef A as Object;
  ...
  public class List extends Collection { ... }
}
```

Here it is essential that `List` be an inner class of `Collection`, so that it inherits use of the virtual type `A`. Now one can declare, say, collections of strings.

```
public class StringCollection extends Collection{
  typedef A as String
}
```

And now `StringCollection.List` is a subtype of `StringCollection`, as desired. However, this approach requires that the designer of collections has the foresight to include lists. A crucial flexibility of object-oriented languages, that one may define lists long after collections, has been lost.

3 Strengths of virtual types

The previous examples focussed attention on strengths of parametric types and weaknesses of virtual types. Here is a second example, aimed to focus attention on weaknesses of parametric types and strengths of virtual types: families of classes.

Palsberg and Schwartzbach [PS94] tell a compelling story about a common situation where families of classes arise. We will repeat that story here, and then abstract from it to give two examples, 'lists with lengths' and 'alternating lists with lengths'. (The examples are closely related to the 'lists with lengths' and XY-grammar examples of Palsberg and Schwartzbach.)

Consider a processor for a programming language. An early phase of the processor uses abstract syntax trees. These are represented by a family of classes: one class for each non-terminal of the grammar. (Each of these classes might in turn have subclasses to represent each alternative production for the non-terminal.) A later phase of the processor uses annotated abstract syntax trees (they might be annotated with types, flow analysis information, or the like). These are represented by a second family of classes, each original class being subclassed to add the annotation. There might be multiple families, one for each phase.

The simplest possible abstraction of this idea is based on the grammar:

```
X ::= char X | empty
```

which corresponds to lists of characters. The inherited class is augmented with a simple annotation, the length of the list.

The next simplest abstraction is based on the grammar

```
X ::= char Y | empty
Y ::= float X | empty
```

which corresponds to lists that consist alternately of characters and floats. Again, the inherited classes augment each list with its length.

Although these examples are simple, the analogy with grammars and annotations should make it clear that they correspond to a range of examples of interest.

Here is the second example, rendered with virtual types.

```
public class XList {
  deftype YThis as YList;
  protected char h;
  protected YThis t;
  public XList (char h, YThis t) {
    super(); setHead(h); setTail(t);
  }
  public char head () { return h; }
  public YThis tail () { return t; }
  public void setHead (char h) { this.h = h; }
  public void setTail (YThis t) { this.t = t; }
}

public class YList {
  deftype XThis as XList;
  protected  float h;
  protected XThis t;
  public YList (float h, XThis t) {
    super(); setHead(h); setTail(t);
  }
  public float head () { return h; }
  public XThis tail () { return t; }
  public void setHead (float h) { this.h = h; }
  public void setTail (XThis t) { this.t = t; }
}

public class LenXList extend XList {
  deftype YThis as LenYList;
  protected int l;
  public LenXList (char h, YThis t) {
    super(h,t);
  }
  public void setTail (YThis t) {
    super.setTail(t);
    if (t == null) l = 1; else l = 1+t.length();
  }
  public int length () { return l; }
}

public class LenYList extend YList {
  deftype XThis as LenXList;
  protected int l;
  public LenYList (float h, XThis t) {
    super(h,t);
  }
  public void setTail (XThis t) {
    super.setTail(t);
    if (t == null) l = 1; else l = 1+t.length();
  }
  public int length () { return l; }
}

public class Breakit {
  public void breakit () {
    XList xl = new LenXList('a', null);
    YList yl = new YList(3.142f, null);
    xl.setTail(yl);  // fails at run-time
  }
}
```

Observe that `XThis` and `YThis`, rather than `XList` and `YList`, are used for the types of the list tails. To demonstrate why covariance (and so-called *binary methods*) are difficult, a `breakit` method has been included. In the absence of suitable run-time or compile-time checks, the `breakit` code would violate the invariant that the tail of a list with length is itself a list with length. In Thorup's version of virtual types, the call labeled above signals a run-time error.

It's a frustrating exercise to attempt to express the same information with parametric types. Here is an attempt to do so. The translation is presented mainly because it offers some interesting insights, as it is too complex to use in practice.

Each class is paired with a 'functor' class (indicated by the suffix `F`), which takes one parameter for each virtual type. The actual class is derived by instantiating the parameters appropriately.

This translation gives `extends` an (arguably counterintuitive) meaning, in that the extended class is a subclass of the generator, not the named class. This sad complication does carry with it a happy benefit, as it causes the `breakit` method to fail at compile-time rather than run-time: one cannot assign a `LenXList` to an `XList`, since the former no longer extends the latter.

In the following the bodies are omitted since they are identical to the initial example (save that constructor names are changed as appropriate, so that the constructor `XList` becomes `XListF` in that class, and so on). The class `XList` is a fixpoint of the functor `XListF`, and so on. The fixpoint classes only contain constructor definitions, which are required since they cannot be inherited.

```
public class XListF
  <XThis extends XListF<XThis,YThis>,
   YThis extends YListF<Ythis,XThis>>
```

```
{ ... }

public class YListF
  <YThis extends YListF<YThis,XThis>,
   XThis extends XListF<XThis,YThis>>
{ ... }

public class XList extends XListF<XList,YList> {
  XList (char h, YList t) { super(h, t); }
}

public class YList extends YListF<YList,XList> {
  YList (char h, YList t) { super(h, t); }
}

public class LenXListF
  <XThis extends LenXListF<XThis,YThis>,
   YThis extends LenYListF<YThis,XThis>>
  extends XListF<XThis,YThis>
{ ... }

public class LenYListF
  <YThis extends LenYListF<YThis,XThis>,
   XThis extends LenXListF<XThis,YThis>>
  extends YListF<YThis,XThis>
{ ... }

public class LenXList
    extends LenXListF<LenXList,LenYList> {
  LenXList (char h, LenYList t) { super(h, t); }
}

public class LenYList
    extends LenYListF<LenYList,LenXList> {
  LenYList (char h, LenXList t) { super(h, t); }
}

public class Breakit {
  public void breakit () {
    XList xl = new LenXList('a', null); // type error
    YList yl = new YList(3.142, null);
    xl.setTail(yl);
  }
}
```

Now the `breakit` method fails at compile time rather than run time. The marked line is a type error because `LenXList` is no longer a subtype of `XList`.

If the above code referred to `this`, an additional trick would be required to give `this` the appropriate type (namely, `XThis` within `XList`, and `YThis` within `YList`). Replace `this` by calls to a method `this.asXThis()`, and add to `XList$gen` the abstract method declaration

```
abstract XThis asXThis();
```

and then add to `XList` the method body

```
public XList asXThis() { return this; }
```

and similarly for `LenXList`. The types work out because `XThis` is instantiated to `XList` in `XList`.

This translation of virtual types into parameterised types closely resembles the usual semantics of object-oriented features in terms of F-bounded polymorphism [CCHOM89, Bru94].

4 Providing virtual types statically

In this section of the paper we present a proposal for extending Java with a construct which is similar to virtual types, but which can be statically type-checked with rules which can be easily understood by programmers. In particular we show that if $I_1, \ldots, I_n$ is a system of interfaces whose definitions are mutually referential (i.e., each may refer to any of the others), then it is possible to define (in a type-safe way) a system of subinterfaces, $J_1, \ldots, J_n$ of the I_i such that all inherited references to the I_i's behave as references to J_i's in the subinterfaces. Moreover classes implementing these interfaces will also have the signatures of methods and fields change appropriately in subclasses implementing the subinterfaces. An important feature of this construction is that classes defined in this way can be statically type-checked.

4.1 The simple case: Understanding `ThisType`

In earlier work [BSvG95, BFP97, Bru97] the authors solved a similar problem with *self*-referential interfaces and classes using a `MyType` or `ThisType` construct. In an interface or class definition, `ThisType` stands for an interface of the object being declared or defined. The following example illustrates the use of `ThisType` in the context of a list whose head is a character and whose tail is a List.

```
interface ListIfc {
  public char head ();
  public @ThisType tail ();
  public void setHead (char h);
  public void setTail (@ThisType t);
}

class List implements @ListIfc {
  protected char h;
  protected @ThisType t;
  public List (char h, @ThisType t) {
    super(); setHead(h); setTail(t);
  }
  public char head () { return h; }
  public @ThisType tail () { return t; }
  public void setHead (char h) { this.h=h; }
  public void setTail (@ThisType t) { this.t=t; }
}
```

Here, within `ListIfc` the type variable `ThisType` stands for the type of `this` for any class that implements the interface `ListIfc`, and within `List` the type variable `ThisType` stands for the type of `this` for any class that implements the public interface of `List`, which is exactly `ListIfc`.

As a first approximation, one might think of `ThisType` as another name for `ListIfc`. However its meaning is a bit more subtle. Just as the meaning of `this` changes in the move to subclasses, the meaning of `ThisType`, which represents the interface of `this`, changes as well. Thus in `List`, all methods are type-checked under the assumption that `this` has interface `@ThisType` and that `ThisType` extends `ListIfc`.

The prefix `@` indicates an "exact" type. That is, if a variable `a_exact` has type `@A`, then values stored in `a_exact` must be of exactly that type, not from an extension. Thus if `B` extends `A`, `a` has type `A`, and `b` has type `B`, then the assignment `a = b` is legal because a variable of type `A` can hold values from any type extending it. However if `a_exact` has type `@A`, then the assignment `a_exact = b` will be illegal since only values of exactly type `A` can be held in `a_exact`. Exact types are needed in handling *binary methods* [BCC+95], *i.e.*, methods with a parameter of type `ThisType`. See Appendix A and either [Bru97] or [BFP97] for more detailed explanations of the use of exact types with *binary methods*.

The advantage in using `ThisType` comes when we define extensions of a class. We can define an extension of `List` which keeps track of the length of the list.

```
public interface LenListIfc extends ListIfc {
  int length();
}

public class LenList extends List
  implements @LenListIfc {
  protected int l;
  public LenList (char h, ThisType t) {
    super(h, t);
  }
  public void setTail (@ThisType t) {
    super.setTail(t);
    if (t == null) l = 1; else l = 1+t.length();
  }
  public int length() { return l; }
}
```

Because the instance variable `t` has type `@ThisType` (rather than `List` or `ListIfc`), when it is inherited its meaning changes along with the meaning of `this`. Thus an object generated by `List`, will have a protected instance variable `t` holding a value of type `@ListIfc`, while the corresponding `t` in an object generated by `LenList` will hold a value of type `@LenListIfc`. Similarly the value returned by sending the `tail()` message to a value generated by `List` will be of type `@ListIfc`, while the result returned will be of type `@LenListIfc` if the same message is sent to a value generated from `LenList`.

Thus the use of `ThisType` provides greater flexibility when defining classes with fields or methods whose signatures should be the same as the object.

We have chosen to have `ThisType` designate an interface rather than a class. Suppose instead that we had included a `ThisClass` construct. Here is a reworking of the previous example using this construct.

```
interface ListClassIfc {
  public char head ();
  public @ThisClass tail ();
  public void setHead (char h);
  public void setTail (@ThisClass t);
}

class ListClass implements ListClassIfc {
  protected char h;
  protected @ThisClass t;
  public List (char h, @ThisClass t) {
    super(); setHead(h); setTail(t);
  }
  public char head () { return h; }
  public @ThisClass tail () { return t; }
  public void setHead (char h) { this.h=h; }
  public void setTail (@ThisClass t) { this.t=t; }
}
```

This greatly reduces flexibility, since we cannot utilize a variable with the interface type `ListClassIfc`. Suppose `xl` has type `ListClassIfc`, and consider the expression `xl.setTail(yl)`. What type should the parameter `yl` have? Intuitively it should be generated by the same class as `xl`, but we cannot tell statically what class generated `xl`. Hence we cannot statically type check this construct. For this reason, we choose for `ThisType` to range over interfaces rather than classes.

4.2 Generalizing `ThisType` to mutually recursive interfaces and classes

We wish to provide a statically checkable language construct to define mutually referential classes, where the classes (and their interfaces) are expected to change in parallel in extensions. Since they refer to each other, we need some way of grouping them together and providing the same sort of flexible names as `ThisType`. In particular, we need a way of associating these flexible names with the appropriate classes.

The only syntactic extensions to Java that we need are two new keywords, `as` and `use`, as well as clauses which use these new keywords to associate flexible names for classes. We associate a `ThisType`-style name to an interface with the following notation.

```
interface Ifc as TT { ... }
```

Here `Ifc` is the name of the interface, while `TT` is a type variable used analogously to the `ThisType` interface, which automatically changes in extensions of the interface. We will designate the name following the keyword "`interface`" as the *specific* name of the interface and the name following the keyword "`as`" as its *variable* name.

We warn the reader that we have interchanged the order of the variable and specific names in the "`as`" clause from that used in the virtual types proposal. The concrete syntax is still under discussion and may change in the future.

We can rewrite class `List` above using the "`as`" construct.

```
public interface ListIfc as TType {
  public char head ();
  public @TType tail ();
  public void setHead (char h);
  public void setTail (@TType t);
}

public class List as TType implements @ListIfc {
  protected char h;
  protected @TType t;
  public List (char h, @TType t) {
    super(); setHead(h); setTail(t);
  }
  public char head () { return h; }
  public @TType tail () { return t; }
  public void setHead (char h) { this.h=h; }
  public void setTail (@TType t) { this.t=t; }
}
```

The "`as`" clause which introduces the variable name `TType` in the declaration of the interface `ListIfc` indicates that `TType` is to be used as the interface of `this` in classes which implement that interface. The only changes made to the class `List` were to include the `implements` clause associating it with the interface and to replace all occurrences of the `ThisType` with the variable name `TType` from `ListIfc`.

Mutually referential interfaces and classes which need to refer to each other's `ThisType` will be grouped together so as to delimit the scope of the definitions. Fortunately, Java 1.1 now includes "inner" or nested interfaces and classes which will provide the framework for this grouping. (Thanks to Kresten Thorup for suggesting this use of inner classes.)

The following is a reworking of the alternating list example in the new notation.

```
public interface AltListGrpIfc {
  public interface XListIfc as XThis {
    char head ();
    @YThis tail ();
    void setHead (char h);
    void setTail (@YThis t);
  }
  public interface YListIfc as YThis {
    float head ();
    @XThis tail ();
    void setHead (float h);
    void setTail (@XThis t);
  }
}

public class AltListGrp implements AltListGrpIfc {
  public static class XList as XThis
    implements @XListIfc
  {
    protected char h;
    protected @YThis t;
    public XList (char h, @YThis t) {
      super(); setHead(h); setTail(t); }
    public char head () { return h; }
    public @YThis tail () { return t; }
    public void setHead (char h) { this.h=h; }
    public void setTail (@YThis t) { this.t=t; }
  }
  public static class YList as YThis
    implements @YListIfc
  {
    protected float h;
    protected @XThis t;
    public YList (float h, @XThis t) {
      super(); setHead(h); setTail(t);
    }
    public float head () { return h; }
    public @XThis tail () { return t; }
    public void setHead (float h) { this.h=h; }
    public void setTail (@XThis t) { this.t=t; }
  }
}
```

The intention is that any implementation of the "outer" interface, `AltListGrpIfc` must provide implementations of the "inner" interfaces, `XListIfc` and `YListIfc`. Because the "outer" interface `AltListGrpIfc` delimits the scope of the `XThis` and `YThis` type variables, we do not need to include a definition of `YThis` in `XListIfc` or of `XThis` in `YListIfc`.

We also make a minor extension to Java, in that we assume that since `AltListGrp` implements `AltListGrpIfc` that the names `XListIfc` and `YListIfc` can be used without qualification inside `AltListGrp`. (Java proper requires one to write `AltListGrp.XListIfc` and `AltListGrp.YListIfc` instead.)

Type checking of the classes is done as follows. Inside `XList`, `this` has type `@XThis`, while inside `YList`, `this` has type `@YThis`. Because each class and interface may be replaced by extensions in a subclass of `AltListGrp`, we only assume that `XThis` extends `XListIfc` and `YThis` extends `YListIfc` when type-checking the classes. We emphasize that the constraints on both `XThis` and `YThis` are available when type-checking each of `XList` and `YList`.

We can now define the subinterfaces and subclasses.

```
public interface LenAltListGrpIfc
  extends AltListGrpIfc
{
  public interface LenXListIfc as XThis
    extends XListIfc
  {
    public int length();
  }
  public interface LenYListIfc as YThis
    extends YListIfc
  {
    public int length();
  }
}

public class LenAltListGp
  extends AltListGp
{
  public static class LenXList as XThis
    extends XList implements @LenXListIfc
  {
    protected int l;
    public LenXList (char h, @YThis t) {
      super(h, t);
    }
    public void setTail (@YThis t) {
      super.setTail(t);
      if (t == null) l = 1; else l = 1+t.length();
    }
    public int length () { return l; }
  }
  public static class LenYList as YThis
    extends YList implements @LenYListIfc
  {
    protected int l;
    public LenYList (float h, @XThis t) {
      super(h,t);
    }
    public void setTail (@XThis t) {
      super.setTail(t);
      if (t == null) l = 1; else l = 1+t.length();
    }
    public int length () { return l; }
  }
}
```

The subinterface `LenAltListGrpIfc` contains extensions of the interfaces `XListIfc` and `YListIfc`, while the subclass `LenAltListGrp` contains extensions of the classes `XList` and `YList` from `AltListGrp`. This time both classes are type checked assuming that `XThis` extends `LenXListIfc` and `YThis` extends `LenYListIfc`. If a new interface redefining a variable interface name is not provided in an extension of the outer class, the default is that that the old one is inherited unchanged. For example, if an interface naming `XThis` in its "`as`" clause were not included in `LenAltListGrpIfc`, then uses of `XThis` inside the "outer inteface" would be interpreted as being tied to `XListIfc`.

We access the inner classes and interfaces by qualifying their names with the enclosing class.

```
public class Useit {
```

```
    public void useit () {
      @AltListGrpIfc.XThis xl
        = new AltListGrp.XList('a',null);
      @AltListGrpIfc.YThis yl
        = new AltListGrp.YList('b',null);
      xl.setTail(yl);
    }
  }
```

When using qualified names, we can choose to use either the specific or variable name of the interface. For maximum flexibility (and in anticipation of the next section) we have chosen to use the latter (*e.g.*, `XThis` rather than `XListIfc`) in this example.

The example in Appendix B also includes a "`use` clause" which can be used to bring externally-defined interfaces and classes into a collection so that extensions can also depend on extensions of those declarations. While this will likely not be used regularly, there are times when it is very helpful to bring in externally defined items.

It will likely be more useful to use variable names (like `YThis`), rather than the specific names (like `YList`). Specific names must be used in `new` expressions, since classes but not interfaces can create new instances. There may also be circumstances in which the specific names are used because the programmer does not wish an interface specification to change in extensions. Thus if we wished the method `setTail` to take a parameter of type `YListIfc` in all subclasses (rather than having it change with the subclass) then we could write the `YListIfc` rather that `YThis` in its declaration.

5 Parametric polymorphism with mutually recursive groups

These constructs interact smoothly with parametric polymorphism, both in defining polymorphic inner classes, and in using the "outer" class as a constraint for type variables. For example we could change our implementation of `AltListGrp` to hold values of a type specified by a type parameter.

```
  public interface PolyAltListGrpIfc
    <XType extends Object, YType extends Object>
  {
    public interface XListIfc as XThis {
      public XType head ();
      public @YThis tail ();
      public void setHead (XType h);
      public void setTail (@YThis t);
    }
    public interface YListIfc as YThis {
      public YType head ();
      public @XThis tail ();
      public void setHead (YType h);
      public void setTail (@XThis t);
    }
  }

  public class PolyAltListGrp
    <XType extends Object, YType extends Object>
    implements PolyAltListGrpIfc<XType,YType>
  {
    public static class XList as XThis
      implements XListIfc
    {
      protected XType h;
      protected YThis t;
      ...
    }
    public static class YList as YThis
      implements YListIfc
    {
      protected YType h;
      protected XThis t;
      ...
    }
  }
```

The previous `AltListGrp.XList` is equivalent to `PolyAltListGrp<char,float>.XList`, and similarly for `YList`.

A different use of polymorphism arises when we use an "outer" interface as a type constraint.

```
  public class Useit<T extends AltListGrpIfc> {
    ... T.XThis ... T.YThis ...
  }
```

Because the type variable `T` extends `AltListGrpIfc`, it must support inner interfaces with variable names `XThis` and `YThis`. These inner interfaces are mutually referential and extend `AltListGrpIfc.XThis` and `AltListGrpIfc.YThis`, respectively. For example, an expression with type `@T.XThis` can be sent a `setTail` message with a parameter of type `@T.YThis`. As a result, groups of interfaces (packaged as inner interfaces) can be passed around and used polymorphically. This extension would be simple to accommodate in proposals where parametric types are implemented by expansion into specialised instances, as in the proposal of Agesen *et.al.* [AFM97] or in the heterogeneous translation of Pizza [OW97]. It is not clear how to implement the extension in a homogeneous translation where type parameters are erased.

6 Conclusion

Parametric types and virtual types each do things well that the other does poorly. Both parametric types and virtual types have been given semantics expressed as translations from extensions to Java into Java as it stands (see [Tho97, OW97]), and these translations look remarkably similar.

While an extension of Java supporting parametric types can be statically typed (see [OW97, Bru97, AFM97, MBL97], for example), Thorup's [Tho97] proposed extension supporting virtual types requires dynamic type checking to preserve type safety. One can translate virtual types into parametric types in a type-safe way, as shown in Section 3, but the result is too cumbersome to be useful in practice.

One way forward is to tack features that mimic parametric types onto virtual types (as proposed by Cartwright and Steele [CS97]), or to tack features that mimic virtual types onto parametric types as proposed here in section 3.

In sections 4 and 5 we present a proposal for such an extension. It supports parametric types directly, and supports virtual types by using inner interfaces and classes to group together mutually recursive declarations that may be changed simultaneously. An important advantage of this language design is is that all type checking can be done statically – it does not require extra dynamic type checking.

A statically safe method to model virtual types was proposed by Torgersen [Tor98]. His work is similar to, but independent from ours. Where we write `interface I as IThis`, Torgersen would declare `I` as an inner interface in a wrapper class which also contains the virtual type declaration `typedef IThis as I`. This is strikingly similar to our treatment of mutually recursive classes. Instead of relying on exact types, Torgersen requires that a class containing a virtual type is declared `final` (so it cannot be redeclared in subclasses) before it can be used as an ordinary type. His scheme has the advantage of needing less machinery, but some inheritance hierarchies cannot be expressed using his technique.

Our notation can be considered as a variant of the proposal for handling inheritance in the presence of mutual recursion put forward by Palsberg and Schwartzbach [PS94]. It differs from their proposal by requiring the interaction of inheritance with recursion to be explicitly declared, which is interesting because they claimed such an option was not possible.

Does our proposal constitute the ultimate solution to integrating parametric and virtual types? We think we've made a useful step, but further study is required.

References

[AFM97] Ole Agesen, Stephen Freund, and John C. Mitchell. Adding parameterized types to Java. In *Symposium on Object-Oriented Programming: Systems, Languages, and Applications*, ACM, 1997.

[BCC+95] Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, The Hopkins Objects Group, Gary T. Leavens, and Benjamin Pierce. On binary methods. *Theory and Practice of Object-Oriented Systems*, 1(3): 221–242,1995.

[BFP97] Kim B. Bruce, Adrian Fiech, and Leaf Petersen. Subtyping is not a good "match" for object-oriented languages. In *ECOOP '97*, pages 104–127. LNCS 1241, Springer-Verlag, 1997.

[Bru94] Kim B.Bruce. A paradigmatic object-oriented programming language: design, static typing and semantics. *Journal of Functional Programming*, 4(2):127–206, 1994.

[Bru97] Kim B. Bruce. Increasing Java's expressiveness with ThisType and match-bounded polymorphism. Technical report, Williams College, 1997. Available via `<http://www.cs.williams.edu/~kim/README.html>`.

[Bru97b] Kim B. Bruce. Safe static type checking with systems of mutually recursive classes and inheritance. Technical report, Williams College, 1997.

[BSvG95] Kim B. Bruce, Angela Schuett, and Robert van Gent. PolyTOIL: A type-safe polymorphic object-oriented language, extended abstract. In *European Conference on Object-Oriented Programming*, pages 27–51, LNCS 952, Springer-Verlang, 1995.

[CCHOM89] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell, F-bounded polymorphism for object-oriented programming. In *Conference on Functional Programming Languages and Computer Architecture*, pages 273–280, ACM, 1989.

[CS97] Corky Cartwright and Guy Steele. Yet another parametric types proposal. Message to Java genericity mailing list, August, 1997.

[GRJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[KM96] Andrew Koenig and Barbara Moo. *Ruminations on C++*. Addison-Wesley, 1996.

[MBL97] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parameterized types for Java. In *Symposium on Principles of Programming Languages*, pages 132–145, ACM, 1997.

[PS94] Jens Palsberg and Michael I. Schwartzbach. *Object-Oriented Type Systems*. Wiley, 1994.

[Rey85] J. C. Reynolds, Three approaches to type structure. In *Mathematical Foundations of Software Development*, LNCS 185, Springer-Verlag, 1985.

[OW97] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Symposium on Principles of Programming Languages*, pages 146–159, ACM, 1997.

[Tho97] Kresten Krab Thorup. Genericity in Java with virtual types. In *European Conference on Object-Oriented Programming*, pages 444–471, LNCS 1241, Springer-Verlag, 1997.

[Tor98] Mads Togersen. Virtual types are statically safe. *5th Workshop on Foundations of Object-Oriented Languages*, January 1998.

A Exact Types

Exact types are necessary when sending "binary" messages to objects. A binary method is one which has a parameter of type `ThisType` in the Java extension proposed in [Bru97]. The method `setTail` in `List` or `ListIfc` is an example of a binary method. The receiver of a binary message (*i.e.*, a message corresponding to a binary method) must be an exact type.

Suppose not. Let `ExtendedList` extend `List` by adding a new public method `doSomething`, which is not included in `List`, and by redefining the method `setTail` in such a way that it includes sending the message `doSomething` to its argument, `tail`. This will not be a problem since `ExtendedList` is type-checked under the assumption that `ThisType` includes all public methods of `ExtendedList`.

Now let `xl` and `yl` be variables declared to have type `ListIfc`. Thus the values of `xl` and `yl` at run time might have been generated by `List`, `ExtendedList`, or any other class implementing `ListIfc`. Is `xl.setTail(yl)` well-typed?

If `xl` and `yl` both hold values generated by the same class then there will be no problem. However, suppose `xl` holds a value generated from `ExtendedList` and `yl` a value from `List`. Then the version of `setTail` in `ExtendedList` will send the message `doSomething` to `yl` which is not capable of responding to the message. Thus `xl.setTail(yl)` should not be well-typed.

However, if both `xl` and `yl` have type `@ListIfc` then `xl.setTail(yl)` will always be type safe as the code for `setTail` in any class implementing `@ListIfc` will be type checked under the assumption that `ThisType` extends `ListIfc`, so it cannot depend on any methods not contained in `ListIfc`.

The notion of binary methods must be extended for the notation proposed in section 4. In this case, a method of an inner interface is considered "binary" if any of its parameters have a type which is a variable name of any other inner interface of the same outer interface. Thus the methods named `setTail` in `XListIfc` (and `YListIfc`) is binary because it has a parameter of type `YThis` (respectively, `XThis`).

Thus the call of `setTail` below will not statically type check.

```
public class Breakit {
  public void breakit () {
    AltListGpIfc.XThis xl
      = new LenAltListGrp.LenXList('a', null);
    AltListGpIfc.YList yl
      = new LenAltListGrp.LenYList(3.142f, null);
    xl.setTail(yl);  // type error
  }
}
```

However, the following code is type correct since the type of the receiver is an exact type.

```
public class Useit {
  public void useit () {
    @AltListGpIfc.XThis xl
      = new AltListGroup.XList('a', null);
    @AltListGpIfc.YThis yl
      = new AltListGroup.YList(3.142f, null);
    xl.setTail(yl);  // now type correct
  }
}
```

Because of space limitations we are unable to include all of the type checking rules for this extension, but the reader is referred to [BFP97] for the typing rules for a similar language, but in which the conventions for exact and "flexible" types are reversed.

B The subject-observer example

In this section we examine another example of the use of virtual types and demonstrate how this would be written in the Java extension suggested in the paper. Thorup's paper [Tho97] used the following example based on the Observer pattern from [GRJV94].

```
public class Observer {
  typedef SType as Subject
  typedef EventType as Object;
  public void notify (SType s, EventType e) {...}
}

public class Subject {
  typedef OType as Observer;
  typedef EventType as Object;
  protected OType observers[];
  ...
  public void notifyObservers(EventType e) {
    int n = observers.length;
    for (int i = 0; i < n; i++)
      observers[i].notify(this,e);
  }
}

public class WindowObserver extends Observer {
  typedef SType as WindowSubject;
  typedef EventType as WindowEvent;
}

public class WindowSubject extends Subject {
  typedef OType as WindowObserver;
  typedef EventType as WindowEvent;
  ...
}
```

The following is a reworking of Thorup's example in our notation.

```
public interface SubObsGrpIfc{
  public interface ObserverIfc as OType {
    void notify (SType s, EventType e);
  }
  public interface SubjectIfc as SType {
    ...
    void notifyObservers (EventType e);
  }
  public use Object as EventType;
}

public class SubObsGrp implements SubObsGrpIfc {
  public static class Observer as OType
    implements @ObserverIfc
  {
    public void notify (SType s, EventType e) {...}
  }
  public static class Subject as SType
    implements @SubjectIfc
  {
    protected @ThisObserver observers[];
    ...
    public void notifyObservers(EventType e){
      int n = observers.length;
      for (int i = 0; i < n; i++)
        observers[i].notify(this,e);
    }
  }
}
```

The only feature not previously introduced is the "`use`" clause in the definition of `SubObsGrpIfc`. This indicates that the type `EventType` will be allowed to vary in extensions of `SubObsGrpIfc`, and hence that methods of `SubObsGrp` should be type checked under only under the assumption that `EventType` extends `Object`.

By our earlier remarks, the occurrence of `this` in the body of `notifyObservers` is type-checked with type `@ThisSubject`.

```
public interface WindowSubObsGrpIfc
  extends SubObsGrpIfc
{
  public interface WindowObserverIfc as OType
    extends ObserverIfc
  { ... }
```

```
  public interface WindowSubjectIfc as SType
    extends SubjectIfc
  {...}
  public use WindowEvent as EventType;
}

public class WindowSubObsGrp extends SubObsGrp
  implements WindowSubObsGrpIfc
{
  public static class WindowObserver as OType
    extends Observer implements WindowObserverIfc
  {...}
  public static class WindowSubject as SType
    extends Subject implements WindowSubjectIfc
  {...}
}
```

The "`use`" clause in `WindowSubObsGrpIfc` introduces `WindowEvent` as the new interpretation for `EventType`. Type-checking rules require that the new interpretation of `EventType` be an extension of the meaning in the super-interface, in this case `Object`. As with the inner interfaces, if no replacement is provided for a variable name specified in a "`use`" clause, the old one is inherited.

Specifying the modification of extended state

K. Rustan M. Leino
Digital Equipment Corporation Systems Research Center
130 Lytton Ave., Palo Alto, CA 94301, U.S.A.
http://www.research.digital.com/SRC/people/Rustan_Leino

Abstract

This paper explores the interpretation of specifications in the context of an object-oriented programming language with subclassing and method overrides. In particular, the paper considers annotations for describing what variables a method may change and the interpretation of these annotations. The paper shows that there is a problem to be solved in the specification of methods whose overrides may modify additional state introduced in subclasses. As a solution to this problem, the paper introduces *data groups*, which enable modular checking and rather naturally capture a programmer's design decisions.

0 Introduction

Specifications help in the documentation of computer programs. Ideally, specifications can be used by a mechanical program analyzer to check the body of a method against its specification, attempting to find errors. The Extended Static Checker, which works on object-oriented programs, is an example of such a program checker [Det96, ESC].

This paper concerns the specification of methods. A method specification is a contract between the implementation of a method and its callers. As such, it includes a *precondition*, which documents what a caller must establish before invoking the method. Consequently, the implementation can assume the precondition on entry to the method body. A method specification also includes a *postcondition*, which documents what the implementation must establish on exit. Consequently, the caller can assume the postcondition upon return from the method invocation. When reasoning about method implementations and calls, only the contract given by the specification is used. That is, one does not use the code in a method's callers when reasoning about the method implementation, and one does not use the implementation when reasoning about the calls.

To be useful to the caller, it is important that the postcondition of a method detail what variables the method does not change. But since the scope of the caller can include variables that are not visible in the scope where the method is declared and specified, it is not possible to explicitly list all unchanged variables in the method's postcondition. Instead, the annotation language must include some form of syntactic shorthand ("sugar") whose interpretation as part of the postcondition is a function of the scope in which it is interpreted. A nice construct for this is the **modifies** clause, which lists those variables that the method is allowed to modify, thereby specifying that the method does not modify any other variables [GH93]. For example, suppose that the specification of a method `m` occurs in a scope where two variables, `x` and `y`, are visible, and that the specification includes the **modifies** clause

modifies `x`

If `m` is called from a scope where, additionally, a variable `z` is visible, then the caller's interpretation ("desugaring") of the specification says that the call may possibly modify `x`, but leaves both `y` and `z` unchanged.

The fact that a **modifies** clause is interpreted differently in different scopes raises a concern about *modular soundness* [Lei95]. For the purpose of this paper, modular soundness means that the implementation, which is checked to meet the specification as interpreted in the scope containing the method body, actually lives up to a caller's expectations, which are based on the specification as interpreted in the scope of the call. A consequence of modular soundness is that one can check a class even in the absence of its future clients and subclasses.

This paper explores the interpretation of specifications in the context of an object-oriented programming language with subclassing and method overrides, for example like Java. In particular, I consider annotations for describing what a method may change and the interpretation of these annotations. I show that there is a problem to be solved in the specification of methods whose overrides may modify additional state introduced in subclasses. As a solution to this problem, I introduce *data groups*, which adhere to modular soundness and rather naturally capture a programmer's design decisions.

For simplicity, I restrict my attention to the operations on only one object, the implicit *self* parameter. Nevertheless, because of inheritance and method overriding, the implementations of the methods of this object may be found in superclasses and subclasses of the class being checked.

1 Extending the state of a superclass

To illustrate the problem, I introduce a simplified example of a computer arcade game—an excellent application of object-

Presented at FOOL'5, 1998

oriented programming indeed.

The design centers around *sprites*. A sprite is a game object that appears somewhere on the screen. In this simple example, every sprite has a position, a color, and methods to update these. The main program, which I will not show, essentially consists of a loop that performs one iteration per video frame. Each iteration works in two phases. The first phase invokes the `update` method on each sprite, which updates the sprite's position, color, and other attributes. The second phase invokes the `draw` method on each sprite, which renders the sprite on the screen.

Here is the declaration of class `Sprite`, in which the methods have been annotated with **modifies** clauses:

```
class Sprite {
  int x, y;
  void updatePosition()  /* modifies x, y */
    { }
  int col;
  void updateColor()     /* modifies col */
    { }
  void update()          /* modifies x, y, col */
    { updatePosition(); updateColor(); }
  void draw()            /* modifies (nothing) */
    { }
}
```

The default `update` method invokes the `updatePosition` and `updateColor` methods, whose default implementations do nothing. Any of these methods can be overridden in `Sprite` subclasses. For example, a moving sprite that never changes colors would override the `updatePosition` method, a stationary sprite whose color changes over time would override the `updateColor` method, and a sprite that adds further attributes that need to be updated overrides the `update` method and possibly also the `updatePosition` and `updateColor` methods.

Since the specifications I have given in the example show only **modifies** clauses, checking that an implementation meets its specification comes down to checking that it modifies only those variables that it is permitted to modify. The implementations of the `updatePosition`, `updateColor`, and `draw` methods are no-ops, so they trivially satisfy their specifications. The `update` method invokes the other two update methods, whose **modifies** clauses say they may modify `x`, `y`, and `col`. So `update` in effect modifies `x`, `y`, and `col`, and this is exactly what its specification allows. We conclude that the methods in class `Sprite` meet their specifications.

Let us now consider a subclass `Hero` of `Sprite`, representing the hero of the game. The hero can move about, and hence the `Hero` class provides its own implementation of the `updatePosition` method by overriding this method. The next position of the hero is calculated from the hero's velocity and acceleration, which are represented as instance variables. The `Hero` class is declared as follows:

```
class Hero extends Sprite {
  int dx, dy;
  int ddx, ddy;
  void updatePosition()
    { x += dx + ddx/2;   y += dy + ddy/2;
      dx += ddx;   dy += ddy;
    }
  ...
}
```

The `Hero` implementation of `updatePosition` increases `x` and `y` by appropriate amounts ($\Delta d = v_0 \cdot t + {}^1/_2 \cdot a \cdot t^2$ where $t = 1$). In addition, it updates the velocity according to the current acceleration. (Omitted from this example is the update of acceleration, which is computed according to the game player's joystick movements.) It seems natural to update the velocity in the method that calculates the new position, but the specification of `updatePosition` (given in class `Sprite`) allows only `x` and `y` to be modified, not `dx` and `dy` which are not even defined in class `Sprite`. (If the update of `dx` and `dy` instead took place in method `update`, there would still be a problem, since the **modifies** clause of `update` also does not include these variables.)

As ecan be seen in this example, the reason for overriding a method is not just to change what the method does algorithmicly, but also to change what data the method updates. In fact, the main reason for designing a subclass is to introduce subclass-specific variables, and it is the uses and updates of such variables that necessitate being able to override methods. For example, class `Sprite` was designed with the intention that subclasses be able to add sprite attributes and update these in appropriate methods. So how does one in a superclass write the specification of a method such that subclasses can extend the superclass's state (that is, introduce additional variables) and override the method to modify this extended state?

2 Three straw man proposals

In this section, I discuss three proposals that I often hear suggested for solving the problem of specifying the modification of extended state. I show that these proposals don't work. This is what it means for a proposal to work:

- the proposal must provide a way to annotate classes like `Sprite` and `Hero` such that the desired method implementations in these classes will meet their specifications,
- the interpretation of specifications must be useful to callers (for example, specifications should not all be treated as "can do anything whatsoever"),
- the annotations should not be unnecessarily tedious to write down, and
- the proposal must adhere to modular soundness.

Here is the first proposal:

> **Straw man 0.** A subclass can *refine* the specification of a method when it overrides it. That is, a subclass can *weaken* the precondition of the method in the superclass (that is, say that the overridden method implementation will work in more situations) and *strengthen* the postcondition (that is, be more specific about the effect of the method).

It is well known that this proposal is sound. However, it doesn't solve the problem at hand. To strengthen the postcondition means to be more precise about the final values of variables. This is just the opposite of what we'd like—we'd like the new postcondition to allow more variables to be modified, that is, to put no restrictions at all on the final values of these variables. Stated differently, while *shrinking* the list in the **modifies** clause is sound, *enlarging* it is what we want when specifying a subclass's method overrides.

Another straw man proposal is the following:

> **Straw man 1.** Let `m` be a method declared and specified in a class `T`. An implementation of `m` is allowed to modify those variables listed in the **modifies** clause of `m`, plus any variable declared in any proper subtype of `T`.

Although sound, this straw man is too liberal about the modification of variables in subclasses. In fact, a subclass loses the advantage of **modifies** clauses with this proposal. To illustrate, I will show an example that builds on class `Sprite`.

Consider a class of *monsters* with a strength attribute. Rather than storing this attribute as an instance variable in every monster object, suppose a class `Monster` has a method that returns the value of the strength attribute. Thus, different `Monster` subclasses can decide on their own representation of the strength attribute. For example, if the strength of a class of monsters is constant, the method can return that constant, without taking up any per-object storage. This design trades quick access of an attribute for flexibility in how the attribute is represented.

The following declaration shows class `Monster`, which uses the strength attribute in updating the sprite position.

```
class Monster extends Sprite {
  int getStrength()  /* modifies (nothing) */
    { return 100; }
  void updatePosition()
    { if (getStrength() < 10) {
          x += 2;
      } else {
          x += 4;
    } }
}
```

A particular `Monster` subclass is `AgingMonster`, which adds an age attribute and overrides the `draw` method so as to render the monster differently according to its strength-to-age ratio.

```
class AgingMonster extends Monster {
  int age;
  ...
  void draw()
    { int bitmapID;
      if (age = 0) {
          bitmapID = MONSTER_INFANT;
      } else {
          int s = getStrength();
          int relativeStrength = s/age;
          if (relativeStrength < 5) {
              bitmapID = MONSTER_WIMPY;
          } elsif (relativeStrength < 10) {
              bitmapID = MONSTER_NORMAL;
          } else {
              bitmapID = MONSTER_STRONG;
      }  }
      Bitmap.Draw(x,y,bitmapID);
    }
}
```

The name `Bitmap.Draw` denotes some procedure that can draw a bitmap given a screen coordinate and an ID.

The correctness of the `AgingMonster` implementation of `draw` relies on the fact that the call to `getStrength` does not modify `age`. In particular, if `getStrength` were to set `age` to `0`, then the computation of `relativeStrength` would result in a division-by-zero error. Method `getStrength` is specified with an empty **modifies** clause, but Straw Man 1 gives implementations of `getStrength` permission to modify `age`, since `age` is declared in a proper subclass of `Monster`. Thus, the interpreted specification for `getStrength` is not strong enough for one to conclude that method `draw` will execute correctly.

There is a workaround. If a class is allowed to refine the specifications of methods declared in superclasses, class `AgingMonster` can strengthen the postcondition of method `getStrength` with $\mathtt{age}_{\mathrm{pre}} = \mathtt{age}_{\mathrm{post}}$. But this would quickly get annoying, because programmers would then sometimes rely on the absence of `age` in the **modifies** clause to conclude that `age` is not changed, and sometimes rely on an explicit postcondition $\mathtt{age}_{\mathrm{pre}} = \mathtt{age}_{\mathrm{post}}$ to conclude the same thing. Even worse, strengthening the specification of all methods declared in a superclass whenever a class introduces new variables would quickly grow to be an unacceptably tedious chore.

The next straw man proposal seeks to alleviate this chore by making the mentioned postcondition strengthening the default interpretation, and providing a new specification construct **also-modifies** that can override the default interpretation:

> **Straw man 2.** Let `m` be a method declared and specified in a class `T`. An implementation of `m` in a subclass `U` of `T` is allowed to modify those variables listed in the **modifies** clause of `m` as given in class `T`, plus any variable declared in any **also-modifies** clause for `m` as given in some superclass of `U`.

This straw man seems to solve the problem for the `Hero` example: One would simply annotate the `updatePosition` override with

also-modifies `dx, dy`

This would give the `updatePosition` implementation in `Hero` permission to modify not just `x` and `y` (as granted by the original specification of `updatePosition` in `Sprite`), but also the variables `dx` and `dy`. (One could also add `ddx` and `ddy` to the **also-modifies** clause, if desired.)

Let us consider how Straw Man 2 stands up to modular soundness. Suppose that the game uses one hero object throughout many game levels. As a new level starts, the program will call a method `startNewLevel` on the hero object. This method resets certain attributes of the hero object while leaving other attributes unchanged, preparing it to begin the new level. To this end, suppose class `Hero` contains the following method declaration and specification, where the keyword **ensures** is used to express a given postcondition:

```
void startNewLevel()
  /*  modifies x, y, col, dx, dy, ddx, ddy
      ensures dx_post = 0 ∧ dy_post = 0     */
  { dx = 0;   dy = 0;
    update();
  }
```

The given implementation of `startNewLevel` contains an error: The invocation of `update` results in a call to the `update` implementation in class `Sprite`, whose invocation of `updatePosition` in turn results in a call to the `updatePosition` implementation in class `Hero` (because of dynamic method dispatch). This implementation of `updatePosition` modifies the `dx` and `dy` variables. Thus, executions of `startNewLevel` may well end with non-zero values for dx and dy, so the implementation of startNewLevel does not meet its specification.

Unfortunately, the methodology proposed by Straw Man 2 does not allow one to catch the error in startNewLevel. The problem is that even though the interpretation of the specification of updatePosition in class Hero reveals that dx and dy may be modified (since the **also-modifies** annotation of updatePosition in class Hero lists these variables), the update method is not overridden in Hero and thus gets its specification solely from the one given in class Sprite. Hence, the interpretation of the specification of update shows dx and dy as being unchanged, so a program checker will not find anything wrong with the implementation of startNewLevel.

Note that the implementations in class Sprite do meet their specifications under Straw Man 2. For example, the interpretation of the specification of updatePosition in class Sprite includes only x and y, both of which are allowed to be modified also by the implementation of update. Hence, there is no error for the checker to report in class Sprite either.

In conclusion, Straw Man 2 seems pretty good at first, but since it allows the specifications of different methods (in the example, updatePosition and update) to be extended in different ways (by having different **also-modifies** clauses, or none at all), the proposal does not adhere to modular soundness. The proposal in the next section provides annotations for data rather than for methods, the effect of which is to make specification extensions apply in a uniform manner.

3 Data groups

In this section, I explain my proposal and demonstrate how it solves the problems with the examples shown previously. In Section 4, I show how a program checker can enforce the proposal, and in Section 5, I argue that my proposal is sound.

The idea is to introduce *data groups*, which represent sets of variables. A data group is declared in a class, just like an instance variable is. The declaration of an instance variable is annotated with the names of the data groups to which the variable belongs. Data groups can be nested, that is, a group can be declared as a member of another group. A data group can be listed in a **modifies** clause, where it represents the set of all members of the group.

Using data groups, the declaration of Sprite can be written as:

```
class Sprite {
  /* group attributes; */
  /* group position  member-of attributes; */
  int x  /* member-of position */;
  int y  /* member-of position */;
  void updatePosition()  /* modifies position */
    { }
  /* group color  member-of attributes; */
  int col  /* member-of color */;
  void updateColor()     /* modifies color */
    { }
  void update()          /* modifies attributes */
    { updatePosition(); updateColor(); }
  /* group drawState; */
  void draw()            /* modifies drawState */
    { }
}
```

This version of class Sprite declares four data groups, attributes, position, color, and drawState, and declares position and color to be members of attributes, x and y to be members of position, and col to be a member of color. Class Sprite does not declare any members of group drawState.

Since updatePosition is declared with the specification **modifies** position, an implementation of this method is allowed to modify x and y. In addition, an implementation of this method is allowed to modify any variables declared in Sprite subclasses to be members of position. An implementation of updatePosition is not allowed to call method updateColor, for example, since color is not a member of position.

By introducing a data group drawState and listing it in the **modifies** clause of method draw, implementations of draw in Sprite subclasses are given a way to modify instance variables (in particular, to modify variables that are introduced as members of drawState).

The following illustrates how one can use data groups to annotate class Hero:

```
class Hero extends Sprite {
  int dx  /* member-of position */;
  int dy  /* member-of position */;
  int ddx  /* member-of position */;
  int ddy  /* member-of position */;
  void updatePosition()
    { x += dx + ddx/2;   y += dy + ddy/2;
      dx += ddx;   dy += ddy;
    }
  void startNewLevel()
    /*  modifies attributes
        ensures dx_post = 0 ∧ dy_post = 0 */
    { dx = 0;   dy = 0;
      update();
    }
}
```

The override of updatePosition gets its permission to modify dx and dy from the fact that these variables are members of the data group position. This solves the problem of how to specify updatePosition in class Sprite so that a subclass like Hero can modify the state it introduces.

With data groups, the error in startNewLevel is detected. Since dx and dy are members of position, which in turn is a member of attributes, a program checker will know that dx and dy may be modified as a result of invoking update. Since the specification of update says nothing further about the final values of dx and dy, one cannot conclude that they remain 0 after the call.

As for the AgingMonster example, the data groups proposal does allow one to infer that no division-by-zero error is incurred in the evaluation of s/age: The guarding **if else** statement guarantees that age is non-zero before the call to getStrength, and since age is not modified by getStrength, whose **modifies** clause is empty, age remains non-zero on return from getStrength.

I will give two more examples that illustrate the use of data groups.

First, note that the members of two groups are allowed to overlap, that is, that a variable is allowed to be a member of several groups. For example, if a Sprite subclass declares a variable

```
int k  /* member-of position,drawState */;
```

then **k** can be modified by any of the methods `update`, `updatePosition`, and `draw`.

Second, I give another example to illustrate that it is useful to allow groups to contain other groups. Suppose a `Sprite` subclass `Centipede` introduces a legs attribute. Class `Centipede` declares a data group `legs` and a method `updateLegs` with license to modify `legs`, which implies the license to modify the members of `legs`. By declaring `legs` as a member of `attributes`, the `update` method gets permission to call method `updateLegs`:

```
class Centipede extends Sprite {
  /* group legs  member-of attributes; */
  int legCount  /* member-of legs */;
  void updateLegs()  /* modifies legs */
    { legCount = ...; }
  void update()
    { updatePosition(); updateColor();
      updateLegs();
    }
}
```

4 Enforcing the data groups proposal

This section describes more precisely how a program checker handles data groups.

For every data group `g`, the checker introduces a new variable `gResidue`. This so-called *residue variable* is used to represent those of `g`'s members that are not in scope—in a modular program, there is always a possibility of a future subclass introducing a new variable as a member of a previously declared group.

To interpret a **modifies** clause

modifies `w`

the checker first replaces `w` with the variables in the *downward closure* of `w`. For any set of variables and data groups `w`, the downward closure of `w`, written `down(w)`, is defined as the smallest superset of `w` such that for any group `g` in `down(w)`, `gResidue` and the variables and groups declared with

member-of g

are also in `down(w)`.

For example, computing the downward closure of the modifies list `attributes` in class `Hero` as shown in Section 3 yields

```
attributes, attributesResidue,
position, positionResidue, x, y, dx, dy, ddx, ddy,
color, colorResidue, col
```

Thus, in that class,

modifies `attributes`

is interpreted as

```
modifies attributesResidue, positionResidue,
         x, y, dx, dy, ddx, ddy,
         colorResidue, col
```

By handling data groups in the way described, the `Hero` implementation of method `startNewLevel`, for example, is allowed to modify `dx` and `dy` and is allowed to call method `update` (but the assignments to `dx` and `dy` must take place *after* the call to `update` in order to establish the specified postcondition of `startNewLevel`). The implementation would also be allowed to call, for example, `updatePosition` directly. The checker would complain if `startNewLevel` called `draw`, because the call to `draw` would be treated as modifying the residue variable `drawStateResidue`, and that variable is not in the downward closure of `attributes`.

5 Soundness

The key to making the data groups proposal sound is that it is always known to which groups a given variable or group belongs, and that residue variables are used to represent members of the group that are not in scope. The data groups proposal is, in fact, a variation of the use of abstract variables and dependencies in my thesis [Lei95]. I will explain the relation between the two approaches in this section, and relegate the proof of soundness to that for dependencies in my thesis.

A data group is like an *abstract variable*. An abstract variable (also called a *specification variable*) is a fictitious variable introduced for the purpose of writing specifications. The value of an abstract variable is represented in terms of program variables and other abstract variables. In some scopes, it is not possible, nor desirable, to specify the representation of an abstract variable because not all of the variables of the representation are visible. This tends to happen often in object-oriented programs, where the representation is often subclass-specific. However, if the abstract variable and *some* of the variables of the representation are visible in a scope, then the fact that there is a dependency between these variables must be known to a program checker in order to achieve modular soundness. Consequently, an annotation language that admits abstract variables must also include some construct by which one can explicitly declare the dependency of an abstract variable on a variable that is part of its representation. For example, if `position` were an abstract variable, then

depends `position` **on** `x`

would declare that variable `x` is part of the representation of `position`. My thesis introduced such dependency declarations. The corresponding notion in this paper is the annotation that declares that `x` is a member of data group `position`:

int `x` /* **member-of** `position` */;

Using dependencies, one can give a precise definition of what the occurrence of an abstract variable in a **modifies** clause means. For dependencies like the ones shown here, this interpretation is the same as that defined for data groups above: the downward closure.

My thesis contains a proof that the use of dependencies in this way adheres to modular soundness, provided the program meets two requirements and provided the interpretation includes residue variables. The two requirements, called the *visibility and authenticity requirements*, together state essentially that a dependency declaration

depends `a` **on** `c`

should be placed near the declaration of `c`. Because the **member-of** annotation is made part of the declaration of the variable whose group membership it declares, the two requirements are automatically satisfied.

There is one other difference between data groups and abstract variables with dependencies. Suppose an abstract variable `a` depends on a variable `c`, and that the downward closure of the **modifies** clause of a method includes `c` but not `a`. The interpretation of such a **modifies** clause says that `c` may be modified, but only in such ways as to not change the abstract value of `a` [Lei95]. This is called a *side effect constraint* on `a`.

But with data groups, it would be meaningless to use side effect constraints, since data groups don't have values. Thus, if variable `c` is a member of a data group `a` and the downward closure of a method `m` includes `c` but not `a`, then the **modifies** clause does not constrain the implementation of `m` in how `c` is changed. Violations of modular soundness result from the deficiency that the different interpretations of a specification in different scopes are inconsistent. So by removing side effect constraints in *all* scopes, modular soundness is preserved.

From our experience with writing specifications for extended static checking, we have found it useful to introduce an abstract variable conventionally called `state` [LN97]. This variable is declared to depend on variables representing the state of a class or module. The `state` variable is used in many **modifies** clauses, but not in pre- and postconditions. Furthermore, `state` is never given an exact definition in terms of its dependencies. Thus, the type of `state` is never important, so we declared its type to be **any**, where **any** is a new keyword that we added to the annotation language.

The data groups proposal grew from a feeling that it was a mistake to apply the side effect constraint on variables like `state` whose type is **any**—after all, the exact value of such a variable is never defined and thus cannot be relied on by any part of the program. By changing the checking methodology to not apply side effect constraints on variables of type **any**, one arrives at the interpretation of data groups presented in this paper.

As a final note on modular soundness, I mention without going into details that the absence of side effect constraints makes the authenticity requirement unnecessary. This means that it would be sound to declare the members of a data group at the time the group is declared, rather than declaring, at the time a variable is declared, of which groups the variable is a member. For example, instead of writing

```
/* group g; */
...
int x  /* member-of g */;
```

one could write

```
int x;
...
/* group g  contains x, ...; */
```

Using **contains** in this way adheres to modular soundness (but declaring a group with both a **contains** and a **member-of** phrase does not). However, while introducing a group containing previously declared variables is sound and may occasionally be convenient, it does not solve the problem described in this paper.

6 Conclusions

In summary, this paper has introduced *data groups* as a natural way to document object-oriented programs. Data groups represent sets of variables and can be mentioned in the **modifies** clauses that document what methods are allowed to modify. The license to modify a data group implies the license to modify the members of the data group as defined by the *downward closure* rule.

Since data groups are closely related to the use of abstract variables and dependencies [Lei95], they adhere to the useful property of *modular soundness*, which implies that one can check a program one class at a time, without needing global program information. Although the literature has dealt extensively with data abstraction and refinement, including Hoare's famous 1972 paper [Hoa72], it seems that only my thesis and my work with Nelson [LN97] have addressed the problem of having abstract variables in **modifies** clauses in a way that modern object-oriented programs tend to use them.

The use of data groups shown in this paper corresponds to *static*, as opposed to *dynamic*, dependencies. Dynamic dependencies arise when one class is implemented in terms of another. Achieving soundness with dynamic dependencies is more difficult than the case for static dependencies [LN97, DLN96].

Data groups can be combined with abstract variables and dependencies. This is useful if one is interested in the abstract values of some attributes and in the representation functions defining these abstract values.

A related methodological approach to structuring the instance variables and methods of a class is *method groups*, first described by Lamping [Lam93] and developed further by Stata [Sta97]. Method groups and data groups both provide ways to organize and think about the variables declared in classes. Other than that, methods groups and data groups have different aims. The aim of method groups is to allow the variables declared in a superclass to be used in a different way in a subclass, a feature achieved by the following discipline: The variables and methods of a class are partitioned into method groups. A variable `x` in a method group `A` is allowed to be modified directly only by the methods in group `A`; methods in other groups can modify `x` only via calls to methods in group `A`. If a designer of a subclass chooses to replace a variable or method of a method group, all variables and methods of the method group must be replaced. The use of method groups can complement the use of data groups, whose aim is to address not *how* variables are used but rather the more fundamental question of *which* variables are allowed to be changed by which methods. If one wants to write specifications in terms of abstract values and allow subclasses to change the representation functions of these abstract values, then one can combine data groups, abstract variables, and dependencies with method groups.

There are many specification languages for documenting object-oriented software, including Larch/C++ [Lea96] and the specification languages surveyed by Lano and Haughton [LH94]. These specification languages do not, however, establish a formal connection between specifications and actual code. Without such a connection, one cannot build a programming tool for finding errors in implementations. As soon as one becomes interested in checking a method implementation against a specification that is useful to callers, one becomes concerned with what the implementation is allowed to modify. Add subclassing to the stew and one faces the problem described in this paper.

To motivate data groups in this paper, I spoke informally about the semantics of the example code. There are several Hoare-like logics and axiomatic semantics of object-oriented programs that define the semantics formally [Lea89, AdB94, Nau94, AL97, Lei97a, PHM97, Lei97b]. Four of these [AL97, Lei97a, PHM97, Lei97b] deal with programs where objects are references to mutable data fields (instance variables) and method invocations are dynamically dispatched. However, except for Ecstatic [Lei97a], these logics have focused more on the axiomatization of language features and object types than on the desugaring of useful specification constructs.

In the grand scheme of annotating object-oriented programs in ways that not only help programmers, but that also can be used by program analyzers, this paper has touched only on the modification of extended state. Though they sometimes seem like a nuisance in the specification of programs, **modifies** clauses are what give a checker precision across procedure boundaries. Vandevoorde has also found **modifies** clauses to be useful in improving program performance [Van94].

Other important method annotations include pre- and postconditions, of which useful variations have also been studied [Jon91, LB97]. As for annotating data, object invariants [Mey88, LW94, LH94, Lea96] is a concept useful to programmers and amenable as annotations accepted by a program checker. Like the modification of extended state, achieving modular soundness with object invariants is an issue [LS97].

Acknowledgements

Raymie Stata, Greg Nelson, Mark Lillibridge, and Martín Abadi made useful comments on a draft of this paper.

References

[AdB94] Pierre America and Frank de Boer. Reasoning about dynamically evolving process structures. *Formal Aspects of Computing*, 6(3):269–316, 1994.

[AL97] Martín Abadi and K. Rustan M. Leino. A logic of object-oriented programs. In Michel Bidoit and Max Dauchet, editors, *Theory and Practice of Software Development: Proceedings / TAPSOFT '97, 7th International Joint Conference CAAP/FASE*, volume 1214 of *Lecture Notes in Computer Science*, pages 682–696. Springer, April 1997.

[Det96] David L. Detlefs. An overview of the Extended Static Checking system. In *Proceedings of The First Workshop on Formal Methods in Software Practice*, pages 1–9. ACM SIGSOFT, January 1996.

[DLN96] David L. Detlefs, K. Rustan M. Leino, and Greg Nelson. Wrestling with rep exposure. Manuscript KRML 68, Digital Equipment Corporation Systems Research Center, July 1996.

[ESC] Extended Static Checking home page, Digital Equipment Corporation Systems Research Center. On the Web at `http://www.research.digital.com/SRC/esc/Esc.html`.

[GH93] John V. Guttag and James J. Horning, editors. *Larch: Languages and Tools for Formal Specification.* Texts and Monographs in Computer Science. Springer-Verlag, 1993. With Stephen J. Garland, Kevin D. Jones, Andrés Modet, and Jeannette M. Wing.

[Hoa72] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–81, 1972.

[Jon91] H. B. M. Jonkers. Upgrading the pre- and postcondition technique. In S. Prehn and W. J. Toetenel, editors, *VDM'91 Formal Software Development Methods, 4th International Symposium of VDM Europe, Volume 1: Conference Proceedings*, volume 551 of *Lecture Notes in Computer Science*, pages 428–456. Springer-Verlag, October 1991.

[Lam93] John Lamping. Typing the specialization interface. *ACM SIGPLAN Notices*, 28(10):201–214, October 1993. OOPSLA '93 conference proceedings.

[LB97] Gary T. Leavens and Albert L. Baker. Enhancing the pre- and postcondition technique for more expressive specifications. Technical Report TR #97-19, Department of Computer Science, Iowa State University, September 1997.

[Lea89] Gary Todd Leavens. *Verifying Object-Oriented Programs that Use Subtypes.* PhD thesis, MIT Laboratory for Computer Science, February 1989. Available as Technical Report MIT/LCS/TR-439.

[Lea96] Gary T. Leavens. An overview of Larch/C++: Behavioral specifications for C++ modules. In Haim Kilov and William Harvey, editors, *Specification of Behavioral Semantics in Object-Oriented Information Modeling*, chapter 8, pages 121–142. Kluwer Academic Publishers, 1996.

[Lei95] K. Rustan M. Leino. *Toward Reliable Modular Programs.* PhD thesis, California Institute of Technology, 1995. Available as Technical Report Caltech-CS-TR-95-03.

[Lei97a] K. Rustan M. Leino. Ecstatic: An object-oriented programming language with an axiomatic semantics. In *The Fourth International Workshop on Foundations of Object-Oriented Languages*, January 1997. Proceedings available from `http://www.cs.indiana.edu/hyplan/pierce/fool/`.

[Lei97b] K. Rustan M. Leino. Recursive object types in a logic of oject-oriented programs. Technical Note 1997-025, Digital Equipment Corporation Systems Research Center, October 1997.

[LH94] Kevin Lano and Howard Haughton, editors. *Object-Oriented Specification Case Studies.* The Object-Oriented Series. Prentice Hall, 1994.

[LN97] K. Rustan M. Leino and Greg Nelson. Abstraction and specification revisited. Manuscript KRML 71, Digital Equipment Corporation Systems Research Center, In preparation, 1997.

[LS97] K. Rustan M. Leino and Raymie Stata. Checking object invariants. Technical Note 1997-007, Digital Equipment Corporation Systems Research Center, April 1997.

[LW94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[Mey88] Bertrand Meyer. *Object-oriented Software Construction.* Series in Computer Science. Prentice-Hall International, New York, 1988.

[Nau94] David A. Naumann. Predicate transformer semantics of an Oberon-like language. In E.-R. Olderog, editor, *Proceedings of the IFIP WG2.1/WG2.2/WG2.3 Working Conference on Programming Concepts, Methods, and Calculi*, pages 467–487. Elsevier, June 1994.

[PHM97] Arnd Poetzsch-Heffter and Peter Müller. A logic for the verification of object-oriented programs. In R. Berghammer and F. Simon, editors, *Programming Languages and Fundamentals of Programming.* Christian Albrechts-Universität Kiel, 1997. Available from `http://voss.fernuni-hagen.de/pi5/forschung/veroeffentlichungen/uebersicht_en.html`.

[Sta97] Raymie Stata. Modularity in the presence of subclassing. Research Report 145, Digital Equipment Corporation Systems Research Center, April 1997.

[Van94] Mark T. Vandevoorde. *Exploiting Specifications to Improve Program Performance.* PhD thesis, Massachusetts Institute of Technology, February 1994. Available as Technical Report MIT/LCS/TR-598.

An Imperative, First-Order Calculus with Object Extension

Viviana Bono
Dipartimento di Informatica
dell'Università di Torino
C.so Svizzera 185,
10149 Torino, Italy
bono@di.unito.it

Kathleen Fisher
AT&T Labs—Research,
180 Park Ave.,
Florham Park, NJ 07932
+1 973 360-8675
kfisher@research.att.com

Abstract

This paper presents an imperative object calculus, designed to support class-based programming via a combination of extensible objects and encapsulation. This calculus simplifies the language presented in [Fis96] in that, like C++ and Java, it chooses to support an imperative semantics instead of method specialization. We show how Java-style classes and "mixins" may be coded in this calculus and prove a subject reduction theorem.

1 Introduction

In the past few years, many theoretical studies have addressed the problem of modeling object-oriented languages via primitive operations [AC96b, FHM94, Fis96, BBL96, PT94, RS98]. Most of these approaches have centered on object-based calculi, where objects, not classes, are seen as the primitive building blocks. However, statically-typed, class-based languages like Simula [BDMN73], Eiffel [Mey92], C++ [ES90, Str94], and Java [AG96] are the most common form of object-oriented language. Hence it is important to evaluate object-based calculi with respect to how well they support class-based programming. The main contribution of this paper is the formal study of a calculus of encapsulated extensible objects, which we use to model class hierarchies. To motivate our study, we begin with a comparison between our approach to modeling classes and the well-known *record-of-premethods* approach. This comparison, which summarizes [FM98], reveals why extensible calculi are relevant to class-based programming. Interested readers are referred to that paper for more details.

1.1 Premethods vs. Extensible Objects

In the context of object calculi, it seems natural to define inheritance using *premethods*, functions that are written with the intent of becoming object methods, but which are not yet installed in any object. Premethods are functions that explicitly depend on the "object itself," typically assumed to be the first parameter to the function. Following this idea, Abadi and Cardelli have proposed encoding classes in a pure object system using records of premethods [AC96c]; these ideas are also used in [RR96]. In this approach, a class is an object that contains a record of premethods and a constructor function used to package these premethods into objects.

The primary advantage of the record-of-premethods encoding is that it does not require a complicated form of object. All that is needed is a way of forming an object from a list of component definitions. However, this approach has some serious drawbacks. We discuss these drawbacks using a list of criteria developed in [FM98] to characterize the role of classes in class-based languages.

Does the class construct provide a coherent, extensible collection? The combination of a record of premethods and a constructor function may be thought of as a coherent, extensible collection. Because premethods are simply fields in a record, nothing requires that they be coherent until a constructor function is supplied. Since the constructor function installs the premethods into an object, however, the fact that a given constructor is typable implies that the premethods it uses are coherent. Notice, however, that nothing requires a given constructor to mention all of the premethods in a given premethod record.

Does the class construct guarantee initialization? In more elaborate record-of-premethod models, the code to initialize private instance variables is guaranteed to run if any of the associated premethods is installed into an object. However, constructor functions cannot be usefully reused in derived classes. A consequence is that if a class designer puts initialization code into a class constructor, that code will not be executed for derived classes. There are several program-development scenarios where this lack would be a serious problem. For example, class designers may wish to perform some kind of bookkeeping whenever objects are instantiated from a class or its descendants. Such bookkeeping is common in database applications, for example. To achieve it, programmers need a place to put code that will execute whenever an object is instantiated. With the record-of-premethods approach, however, there is no appropriate place: no base class constructor function will be called for derived classes, and a premethod function may be called without creating an object.

Does the class construct provide an explicit type hierarchy? In many existing class-based languages, it is possible to restrict the subtypes of an object type to classes that inherit all or part of the class implementation. This restriction may be useful for optimizing operations on objects, allowing access to argument objects in binary methods, and guaranteeing semantic consistency beyond type considerations (some

Presented at FOOL'5, 1998

discussion of these points appears in [KLM94]). A special case of this capability is the ability to define *final* classes, as recognized in work on Rapide [KLM94] and incorporated (presumably independently) as a language feature in Java. This ability is lacking in the record-of-premethods approach since any object whose type is a structural subtype of another type τ can be used as an object of type τ.

Does the class mechanism automatically propagate base class changes? Because derived class constructors must explicitly name the methods that they wish to inherit, the record-of-premethods approach does not automatically propagate base class method changes. In particular, if a derived class `D` is defined from a base class `B` in Java or related languages, then adding a method to `B` will result in an additional method of `D`, and similarly for every other class derived from `B` (and there may be many). With the record-of-premethods approach, derived class constructors must be explicitly rewritten each time base classes change. Since object-oriented programs are typically quite large and maintenance may be distributed across many people, the person who maintains a base class may fail to inform those maintaining its derived classes of its change, causing unpredictable errors. There is no mechanism in this approach to detect such errors automatically.

While we readily admit that simplicity is a virtue, the above discussion reveals that several important and desirable features of class-based programming are lost in the record-of-premethods model. Extensible objects provide a rich alternative. As discussed in [FM98, Fis96], a principled way to think about class-based object-oriented languages is as the combination of two orthogonal components: (i), an object system that supports inheritance and message sending and (ii), an encapsulation mechanism that provides hiding. We call this model of classes the "Classes = Extensible Objects + Encapsulation" approach. Referring to the class-evaluation checklist we used to evaluate the pre-methods model, we can see that this approach successfully addresses each of the points listed there: it provides an extensible coherent collection, guarantees initialization, supports an explicit type hierarchy, and automatically propagates base class changes.

Extensible, Coherent Collection. Extensible objects obviate the need for premethods, since collections of methods that are already installed in objects may be extended. Because of this fact, we may impose static constraints on the ways in which one method may be combined with others. For example, if an object contains two mutually recursive methods, then we cannot replace one with another of a different type. In contrast, in the record-of-premethods approach, it is possible to form a record of premethods without a "covering" constructor that checks to be sure that all of the premethods are coherent.

Guaranteed Initialization. A second advantage of extensible objects is that class constructors and initialization code can be inherited, *i.e.*, reused in derived classes. For example, to create `ColorPoint` objects, we may invoke the `Point` class constructor and add color methods to the resulting extensible object. This process guarantees that the `Point` class has the opportunity to initialize properly any inherited components. It also guarantees that the designers of the `Point` class have the opportunity to update any bookkeeping information they may be keeping about instantiations of `Point` objects.

Explicit Type Hierarchy. A further advantage is the rich subtyping structure of this approach. In particular, it provides "implementation" types that subtype along the inheritance hierarchy and "interface" types that subtype via structural subtyping rules. In addition, it includes a hybrid subtyping relation that allows implementation types to be subtypes of interface types. With this subtyping structure, programmers can use implementation types where the extra information is useful and interface types where more generality is required.

Automatic Propagation of Changes. Another advantage arises with private (or protected) methods. In the extensible-object formulation, methods always remain within an object, even when it is extended. These hidden methods exist in all future extensions, but they can only be accessed by methods that were defined before the method became hidden. Furthermore, these private methods need not be explicitly manipulated by derived class constructors to insure that they are treated properly.

These advantages may be seen in the first example in Section 4, which shows how a `Point-ColorPoint` hierarchy can be formally modeled in our calculus.

1.2 Our Extensible Calculus

Fisher's thesis [Fis96] formally presents a functional object calculus that includes extensible objects and encapsulation. This calculus realizes the benefits described in the previous section. In addition, it includes "mytype" type variables to support method specialization. Unfortunately, these variables greatly complicate analysis. Consequently, to understand the "Classes = Extensible Objects + Encapsulation" idea in its simplest form, Fisher and Mitchell use a simpler language without a "mytype" mechanism in [FM98]. However, they do not study the formal properties of that simpler language. The formal study of such a language is the subject of this paper.

Our calculus, like the one in Fisher's thesis, includes extensible objects and encapsulation; however, it differs in several key aspects. In particular, to compensate for the lack of a "mytype" mechanism, we have adopted an imperative semantics. Hence instead of cascading messages:

$$((\mathtt{ob} \Leftarrow \mathtt{m}_1(\mathtt{arg}_1)\ldots) \Leftarrow \mathtt{m}_n(\mathtt{arg}_n)$$

a programmer can send a sequence of messages to the same object:

$$\mathtt{ob} \Leftarrow \mathtt{m}_1(\mathtt{arg}_1); \quad \cdots; \quad \mathtt{ob} \Leftarrow \mathtt{m}_n(\mathtt{arg}_n);$$

Moreover, in [Fis96], the type system insures the soundness of the object extension operator (a method `m` can only be added to an object that does not already have a method `m`) by tracking method *absences*. Tracking absences requires recording the list of all methods which might be added to the object in the future. This practice has the unfortunate consequence of requiring parent classes to guess what methods their descendants will want to add. In contrast, in the calculus described in this paper, we adjust the type system to track potential method *presences* instead, relieving super objects (*i.e.*, parent classes) of the need for clairvoyance.

To facilitate the type soundness proof, we have added bounded universal types to our language. In the process, we have gained the ability to write "mixins," functions that take an object as an argument and return an extension of that object as a result. For example, a `color` mixin might add a `color` method to any object that does not already contain `color`.

The rest of this paper is structured as follows. In Sections 2 and 3 we formally present our calculus and its type system. In Section 4, we show how this calculus may be used to encode Java-style classes and to model mixins. We introduce our imperative operational semantics in Section 5 and sketch a proof of a subject reduction property in the style of [AC96a] in Section 6. We conclude with a discussion of related work and some thoughts on future directions.

2 The Syntax

The calculus extends an untyped lambda calculus with (*i*) object primitives to build extensible and updatable objects, (*ii*) universal types to write polymorphic method bodies, and (*iii*) existential types to provide encapsulation. Together, these language features may be used to model class hierarchies. Consequently, classes do not appear as primitive expressions in our calculus.

Intuitively, an object is a collection of *methods* and *fields*, each of which has a name and a body. We syntactically distinguish between methods and fields because they have different semantic and typing properties. Briefly, a method is a function which take as an (implicit) argument the object itself, whereas a field can be seen as part of its host object's state. We discuss the distinctions between methods and fields in more detail in Sections 3.3 and 5.

An expression of the untyped calculus can be any of the following:

$$\begin{array}{lll} e & ::= & x \mid c \mid \lambda x.\, e \mid e_1 e_2 \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \mid \\ & & \langle\rangle \mid \\ & & e \Leftarrow m \mid \langle e_1 \leftarrow m = e_2\rangle \mid \langle e_1 \leftarrow\!\!+\, m = e_2\rangle \mid \\ & & e.f \mid \langle e_1.f := e_2\rangle \mid \langle e_1 \leftarrow\!\!+\!: f = e_2\rangle \mid \\ & & \Lambda().\, e \mid e * () \mid \\ & & \{\!\mid r <:_w R :: L = R_1,\ e \mid\!\} \mid \\ & & \mathbf{Abstype}\ r <:_w R :: L\ \mathbf{with}\ x : \tau\ \mathbf{is}\ e_1\ \mathbf{in}\ e_2 \end{array}$$

where x is a term variable, c is a constant, $\lambda x.\, e$ is function abstraction, $e_1 e_2$ is function application, and **let** $x = e_1$ **in** e_2 is let abstraction. In the object-related forms, we use m to denote method names and f for field names. These object forms may be interpreted as follows:

- $\langle\rangle$ is the empty object.
- $e \Leftarrow m$ and $e.f$ are respectively method and field invocation on object e.
- $\langle e_1 \leftarrow m = e_2\rangle$ and $\langle e_1.f := e_2\rangle$ respectively override an existing method or field in e_1 with a new body e_2.
- $\langle e_1 \leftarrow\!\!+\, m = e_2\rangle$ and $\langle e_1 \leftarrow\!\!+\!: f = e_2\rangle$ extend e_1 with a new method or a new field, respectively. In either case, expression e_2 provides the body of the new component.

We defer a discussion about the remaining syntactic forms to Section 3, as they are closely related to types. We also delay the presentation of the operational semantics of our calculus so that we may first present our type system and describe in a reasonably complete fashion how these constructs may be used to model classes. It is hoped that this discussion may develop the reader's intuition for our calculus.

3 The Type System

Our type system is centered around object types. Informally, an object type is a list of *label:type* pairs, each representing a method or field name with its type. Our calculus supports two sorts of object, as in [FM95a]: *prototypes*, typed with **pro**-types, and "proper" objects, typed with **obj**-types. A prototype can be extended with new methods and fields and its components can be overridden, but it supports only trivial subtyping. A prototype can be "sealed" to reach the status of proper object, to which width and depth subtyping applies, but then it is no longer extensible or overridable. This restriction is to avoid the well-known unsoundness that arises when combining object operations with subsumption (see for example [FM95b]). Object types are used in conjunction with existentially quantified types to type encapsulated extensible objects, which we use to model class hierarchies.

The type syntax is as follows:

$$\begin{array}{lll} \text{Types:} & & \\ \quad \tau & ::= & unit \mid \tau_1 \to \tau_2 \mid \mathbf{pro}.R \mid \mathbf{obj}.R \mid \\ & & \forall(r <:_w R :: L)\tau \mid \exists(r <:_w R :: L)\tau \\ \text{Rows:} & & \\ \quad R & ::= & r \mid \langle\!\langle\rangle\!\rangle \mid \langle\!\langle R \mid l : \tau\rangle\!\rangle \\ \text{Kinds:} & & \\ \quad \gamma & ::= & T \mid L \\ \quad All & ::= & \text{Countable set of method and field labels.} \\ \quad L & ::= & \mathrm{L} \subseteq All \end{array}$$

Types include the *unit* type, function types, object types of the two sorts, and universally and existentially quantified types. Rows include row variables r, the empty row $\langle\!\langle\rangle\!\rangle$, and rows formed by extending other rows with a *label:type* pair, $\langle\!\langle R \mid l : \tau\rangle\!\rangle$. Well-formed types have kind T, while rows have kinds that are subsets of All, the set of all labels that can occur in a row. Intuitively, if row R has kind L, then at least all the labels listed in R appear in L. For notational convenience, we will write closed rows (*i.e.*, rows without row variables) using the notations $\langle\!\langle m_i : \tau_i,\ f_j : \tau'_j\rangle\!\rangle^{i \in I, j \in J}$ and $\langle\!\langle l_1 : \tau_1, \ldots, l_n : \tau_n\rangle\!\rangle$.

Rows, introduced in [Wan78], are the tool we use to incorporate polymorphism into our system. We need polymorphism to type extensible objects: because each method takes its host object as an argument, each method body must be polymorphic to work properly in any extensions of its original host. Technical considerations require that method body types be closed to prove our subject reduction theorem. Consequently, we have introduced universal quantifiers on row variables to provide closed, polymorphic types. Fortuitously, this type construct makes our system more expressive, for it allows us to type *mixins*, functions that take objects as parameters and return extensions of those objects as results. Finally, existential quantifiers on row variables produce abstract "object" types. Intuitively, such types reveal an object's interface (*i.e.*, the list of methods and fields to which the object will respond) and abstract away implementation details in a fashion similar to abstract data types.

Because of their roles, we find it useful to have partial information about the row variables that are quantified in $\forall$ and $\exists$ types. To that end, each row variable is declared with a width-supertyping bound R and a kind L: $(r <:_w R :: L)$. Intuitively, the width-bound stands for the minimum amount of information that any instance of the row variable

r may contain. The kind L records an upper bound on the labels present in any instance of r. The labels in L provide the positive information to track method *presences* that we discussed in Section 1: a new label can be added to a row only if it is not already present in L.

With this background, we may now discuss universal and existential quantifier-related expressions. The universal quantifier forms are $\Lambda().\,e$ for row abstraction and $e * ()$ for row application. These forms can be considered as placeholders for the universal quantifier introduction and elimination typing rules. An abstract datatype declaration

$$\textbf{Abstype}\ r <:_w R\ ::L\ \textbf{with}\ x : \tau\ \textbf{is}\ e_1\ \textbf{in}\ e_2$$

declares a data structure named x whose (public) interface is represented by row R and kind L. Intuitively, R describes the components guaranteed to be in x, while L describes the component names that *may* be used in x. Data structure x can only be extended with components whose names are not listed in L. Expression e_1 is the implementation of x, and e_2 represents the code in which this data structure may be used. An implementation ($\{\!|r <:_w R\ ::L = R',\ e|\!\}$) of an abstract type is characterized by R' and e. Row R' is the actual set of components used to implement the data structure. The components listed in both R and R' may be regarded as public, while those appearing only in R' are private. Expression e is the (hidden) implementation code. We use these two constructs to encapsulate the implementations of objects.

3.1 Judgment Forms

Contexts in our calculus are defined as follows:

$$\begin{array}{lcl} \rho & ::= & \epsilon \mid \rho,\,(r <:_w R :: L) \\ \Gamma & ::= & \emptyset \mid \Gamma,\, x : \tau \end{array}$$

We refer to them respectively as *row* and *term* contexts. Functions $dom(\rho)$ and $dom(\Gamma)$ are defined in the standard way. $FV(\Gamma)$ denotes the set of row variables appearing free in any type τ such that $x : \tau \in \Gamma$, for some x. Judgments are as follows:

$\rho \vdash *$	well-formed row context
$\rho \vdash \tau : T$	well-formed type τ
$\rho \vdash R :: L$	row R has kind L
$\rho \vdash \tau_1 <: \tau_2$	type τ_1 subtype of τ_2
$\rho \vdash R_1 <:_B R_2$	row R_1 is a B-subtype of R_2
$\rho\ ;\ \Gamma \vdash *$	well-formed contexts
$\rho\ ;\ \Gamma \vdash e : \tau$	term e has type τ

where the B in the fifth judgment form is a *subtyping annotation* that denotes which subtyping relation has been applied between the two rows R_1 and R_2 (width, or width and depth). We will use the meta-judgment $H \vdash A$ to range over all of the above judgments. In addition, we use the meta-variable U to range over types τ and rows R. The meta-judgment $\rho \vdash U : -\gamma$ represents judgments of the form $\rho \vdash \tau : T$ and $\rho \vdash R :: L$. Similarly, we use meta-judgment $\rho \vdash U_1 <:_{(B)} U_2$ to represent the judgments $\rho \vdash \tau_1 <: \tau_2$ and $\rho \vdash R_1 <:_B R_2$. Finally, we will use the syntax $\textbf{probj.}R$ as shorthand for either $\textbf{pro.}R$ or $\textbf{obj.}R$.

3.2 Subtyping

Our subtyping system includes the standard rules for reflexivity, transitivity, and the contravariant rule for function subtyping. It also makes type *unit* the top type with respect to the subtyping order. In addition, we have a rule for subtyping between object types ($<:$ *obj*):

$$\frac{\rho \vdash R_1 <:_B R_2}{\rho \vdash \textbf{probj.}R_1 <: \textbf{obj.}R_2}$$

This rule is used both to "seal" objects, *i.e.*, promote them from a **pro**-state to an **obj**-state, and to infer subtyping relations between **obj**-types. Subtyping between objects occurs at the level of their component lists, *i.e.*, the rows. For example, we have the following rules to compute width and depth subtyping, respectively:

$$(<: w)\qquad \frac{\rho \vdash R_1 <:_B R_2 \qquad \rho \vdash \langle\!\langle R_1 \,|\, l : \tau \rangle\!\rangle :: L_1}{\rho \vdash \langle\!\langle R_1 \,|\, l : \tau \rangle\!\rangle <:_B R_2}$$

$$(<: d)\qquad \frac{\rho \vdash R_1 <:_B R_2 \qquad \rho \vdash \tau_1 <: \tau_2 \qquad \rho \vdash \langle\!\langle R_i \,|\, m : \tau_i \rangle\!\rangle :: L_i \quad i \in \{1,2\}}{\rho \vdash \langle\!\langle R_1 \,|\, m : \tau_1 \rangle\!\rangle <:_{B+d} \langle\!\langle R_2 \,|\, m : \tau_2 \rangle\!\rangle}$$

Note that the depth subtyping rule applies only to methods. Because fields are updatable, we prevent depth subtyping from applying to fields to preserve type soundness. As this point illustrates, width and depth subtyping have different properties. It is therefore crucial that we control where depth subtyping can occur. For example, when we substitute a row R for an existentially or universally bound row variable r, we need to know that R is a width-only subtype of r's bound. Hence we annotate symbol "$<:$" with B:

$$B\ ::=\ w \mid w,d \mid B_1 + B_2$$

Here, w stands for "width-only" and w,d stands for "depth-allowed". Annotation w is less than w,d, and $+$ denotes the least upper bound of B_1 and B_2 with respect to this ordering. Our subtyping rules on rows are designed to track where depth subtyping occurs using these annotations. The complete set of subtyping rules is in Appendix A.

3.3 Typing Rules

In this section, we discuss the key typing rules of the system. For reference, all typing rules appear in Appendix A, together with the standard rules for inferring the well-formedness of contexts, rows, and types.

We examine the (*meth ext*) rule in detail to illustrate how we use row-polymorphism:

$$\frac{\begin{array}{c}\rho\ ;\ \Gamma \vdash e_1\ :\ \textbf{pro.}R \\ \rho \vdash R :: L \qquad m \notin L \\ \rho\ ;\ \Gamma \vdash e_2 : \forall (r <:_w \langle\!\langle R \,|\, m\ :\ \tau \rangle\!\rangle :: All)\textbf{pro.}r \to \tau\end{array}}{\rho\ ;\ \Gamma \vdash \langle e_1 \leftarrow\!\!+\, m = e_2 \rangle\ :\ \textbf{pro.}\langle\!\langle R \,|\, m : \tau \rangle\!\rangle}$$

Judgment $\rho\ ;\ \Gamma \vdash e_1\ :\ \textbf{pro.}R$ asserts that expression e_1 is a prototype with methods listed in row R. The next judgment insures that row R has at most the labels in L; the side-condition assures us that m is not one of these labels. The final judgment guarantees that method body e_2 is appropriately polymorphic: it will map any possible extension of its current host object, called *self* and typed by $\textbf{pro.}r$, to an expression of type τ, the type of method m. Type $\textbf{pro.}r$

is an appropriate choice to represent the type of *self* since r may be instantiated with any row that has *at least* the methods in $\langle\!\langle R \mid m : \tau \rangle\!\rangle$ and may have *any* other methods, reflected in its maximal kind *All*. Internal extensions (*i.e.*, extensions on *self*) are unsound, so they must be forbidden. The kind *All* in the (*meth ext*) typing rule asserts that *self* may already contain "all" possible methods, so it cannot be extended with any more.

The rule for method override (*meth over*) is similar:

$$\frac{\begin{array}{l}\rho\,;\Gamma \vdash e_1 : \mathbf{pro.}R \\ \rho \vdash R <:_w \langle\!\langle m : \tau \rangle\!\rangle \\ \rho\,;\Gamma \vdash e_2 : \forall(r <:_w R :: All)\mathbf{pro.}r \to \tau\end{array}}{\rho\,;\Gamma \vdash \langle e_1 \leftarrow m{=}e_2\rangle : \mathbf{pro.}R}$$

The only difference between this rule and the previous one is that here the overridden method must be present in the object. The judgment $\rho \vdash R <:_w \langle\!\langle m : \tau \rangle\!\rangle$ guarantees this condition. Note the use of width-only subtyping.

The corresponding rules for fields are (*field ext*) and (*field over*). The differences between these rules and the corresponding ones for methods highlight one of the differences between methods and fields: method bodies are polymorphic, but field bodies are not. This difference arises from the fact that each method body takes its host object (*self*) as an argument so it can operate on that host. In contrast, fields cannot access their host and hence need not be polymorphic. Another difference is that field override on proper objects is sound because depth subtyping is not permitted on fields (In the (*field over*) rule, the first premise allows fields to be overridden in both **pro**- and **obj**-typed objects).

Invocation of a method is typed by rule ($\Leftarrow$ *select*):

$$\frac{\begin{array}{c}\rho\,;\Gamma \vdash e : \mathbf{probj.}R \\ \rho \vdash R <:_w \langle\!\langle m : \tau \rangle\!\rangle\end{array}}{\rho\,;\Gamma \vdash e \Leftarrow m : \tau}$$

The second judgment assures the presence of the method being invoked. The (*field select*) rule is similar.

The universal quantifier introduction rule ($\forall$ *intro*) and existential quantifier elimination rule ($\exists$ *elim*) are classical. The universal quantifier elimination rule ($\forall$ *elim*):

$$\frac{\begin{array}{c}\rho\,;\Gamma \vdash e : \forall(r <:_w R :: L)\tau \\ \rho \vdash R' <:_w R \\ \rho \vdash R' :: L\end{array}}{\rho\,;\Gamma \vdash e * () : [R'/r]\tau}$$

shows that the row variable r can be instantiated with any row that extends its upper bound row and has the proper kind. The existential quantifier introduction rule ($\exists$ *intro*):

$$\frac{\begin{array}{c}\rho \vdash R' <:_w R \\ \rho \vdash R' :: L \\ \rho\,;\Gamma \vdash e : [R'/r]\tau\end{array}}{\rho\,;\Gamma \vdash \{\!|(r <:_w R :: L) = R',\ e|\!\} : \exists(r <:_w R :: L)\tau}$$

matches the semantics of data type implementations: row R' is the private interface of the data structure. It must be consistent with bound R and kind L, as required by the first two hypotheses. The final hypothesis requires expression e to have the appropriate type when r is replaced by R'.

4 Examples

4.1 Class Hierarchy

Using the extensible objects and abstract datatype mechanism from the previous sections, we may encode a simple class hierarchy of `Points` and `ColorPoints` as shown in Figure 1. The two abstract type declarations in the figure define `Point` and `ColorPoint` classes, in that order. The `Point` **Abstype** expression introduces row variable p, which formally serves as the "name" of the `Point` class. Row p is bounded by row P_{pub}, `Point`'s public interface. This row, defined in Figure 1, lists the methods and fields available to clients and descendants of the `Point` class, namely `getX` and `setX`. The kind L_p, defined in Figure 1 to be label set $\{\mathtt{x}, \mathtt{getX}, \mathtt{setX}\}$, indicates that descendant classes may add any methods except ones with these names.

The **with** clause of the declaration specifies that the constructor for the `Point` class, named `newPoint`, will return expressions with type **pro.**p. The constructor returns a prototype so that descendant classes may define their objects via extension from the `Point` class constructor. The row variable p in this return type signals that objects returned by the constructor are defined in the `Point` class. We call such types *partially abstract* after [CW85] because a portion of their structure is hidden.

The **is** clause of the abstraction describes the implementation of the `Point` class. Within this implementation, the row P_{priv} describes the full representation of `Point` class objects by listing all of the components defined in the class and giving their associated types. The expression $\mathtt{Imp}_p$, defined towards the bottom of Figure 1, gives the implementation of the constructor `newPoint`. The type system insures that $\mathtt{Imp}_p$'s type is **pro.**p with p replaced by `Point`'s private interface, P_{priv}. The **Abstype** construct binds row variable p to P_{priv} and variable `newPoint` to $\mathtt{Imp}_p$ within the scope of its **in** clause.

The structure of the `ColorPoint` **Abstype** declaration is similar. There are several aspects of this declaration that are worth mentioning however. In particular, notice that the `ColorPoint` constructor, implemented via the $\mathtt{Imp}_{cp}$ function defined at the bottom of Figure 1, invokes `newPoint` to inherit `Point`'s implementation and initialization code. Notice also that the `ColorPoint` class publicly names the `Point` class as its parent by listing p in its public interface CP_{pub}. Without thus naming its parent, the `ColorPoint` object type (**obj.**cp) would not be a subtype of `Point`'s object type (**obj.**p) because the only way to be a subtype of a partially abstract type is to extend that type. The row variable p in CP_{pub} signals that `ColorPoint` objects were formed via extension from `Point` prototypes, and hence their partially abstract types may be related via subtyping. This fact is the theoretical counterpart to the observation we made in the introduction: a portion of the subtyping hierarchy is determined by inheritance relationships.

Finally, the two **let** declarations in Figure 1 give constructors for the two classes that return proper objects, instead of prototypes. Technically, this shadowing is type correct because **pro** types are subtypes of **obj** types [FM95a].

4.2 Mixins

Because of the bounded universal quantifiers in our calculus, we may type *mixins*, functions that take objects as parameters and return extensions of those objects as results. For example, Figure 2 shows a mixin function `addMove` that adds a `move` method to any object that already has *int* `getX` and

$$
\begin{array}{rl}
\textbf{Abstype} & p <:_w P_{pub} :: L_p \\
\textbf{with} & \texttt{newPoint} : int \rightarrow \textbf{pro.}\, p \\
\textbf{is} & \{\!|\, p <:_w P_{pub} :: L_p = P_{priv},\ \texttt{Imp}_p \,|\!\} \\
\textbf{in} & \\
& \begin{array}{rl}
\textbf{Abstype} & cp <:_w CP_{pub} :: L_{cp} \\
\textbf{with} & \texttt{newCPoint} : int \rightarrow color \rightarrow \textbf{pro.}\, cp \\
\textbf{is} & \{\!|\, cp <:_w CP_{pub} :: L_{cp} = CP_{priv},\ \texttt{Imp}_{cp} \,|\!\} \\
\textbf{in} & \\
& \begin{array}{ll}
\textbf{let} & \texttt{newPoint} : int \rightarrow \textbf{obj.}\, p = \texttt{newPoint} \\
\textbf{let} & \texttt{newCPoint} : int \rightarrow color \rightarrow \textbf{obj.}\, cp = \texttt{newCPoint} \\
\textbf{in} & \langle \texttt{Program} \rangle
\end{array}
\end{array}
\end{array}
$$

where

$$
\begin{array}{lcl}
P_{pub} & \stackrel{def}{=} & \langle\!\langle \texttt{getX} : int,\ \texttt{setX} : int \rightarrow unit \rangle\!\rangle \\
P_{priv} & \stackrel{def}{=} & \langle\!\langle \texttt{x} : int,\ \texttt{getX} : int,\ \texttt{setX} : int \rightarrow unit \rangle\!\rangle \\
CP_{pub} & \stackrel{def}{=} & \langle\!\langle p \mid \texttt{getC} : color,\ \texttt{setC} : color \rightarrow unit \rangle\!\rangle \\
CP_{priv} & \stackrel{def}{=} & \langle\!\langle p \mid \texttt{c} : color,\ \texttt{getC} : color,\ \texttt{setC} : color \rightarrow unit \rangle\!\rangle \\
L_p & \stackrel{def}{=} & \{\texttt{x}, \texttt{getX}, \texttt{setX}\} \\
L_{cp} & \stackrel{def}{=} & \{\texttt{x}, \texttt{getX}, \texttt{setX}, \texttt{c}, \texttt{getC}, \texttt{setC}\}
\end{array}
$$

and

$$
\begin{array}{lcll}
\texttt{Imp}_p & \stackrel{def}{=} & \lambda \texttt{xi}.\langle\ \langle\rangle & \longleftarrow\!\!+\!: \ \texttt{x} = \texttt{xi}; \\
& & & \longleftarrow\!\!+\ \texttt{getX} = \Lambda().\,\lambda\texttt{self}.(\texttt{self.x}); \\
& & & \longleftarrow\!\!+\ \texttt{setX} = \Lambda().\,\lambda\texttt{self}.\,\lambda\texttt{newX}.\ \texttt{self.x}{:=}\,\texttt{newX}\rangle;
\end{array}
$$

$$
\begin{array}{lcll}
\texttt{Imp}_{cp} & \stackrel{def}{=} & \lambda\texttt{xi}.\lambda\texttt{ci}. & \\
& & \langle\, \texttt{newPoint xi} & \longleftarrow\!\!+\!: \ \texttt{c} = \texttt{ci}; \\
& & & \longleftarrow\!\!+\ \texttt{getC} = \Lambda().\,\lambda\texttt{self}.(\texttt{self.c}); \\
& & & \longleftarrow\!\!+\ \texttt{setC} = \Lambda().\,\lambda\texttt{self}.\,\lambda\texttt{newC}. \\
& & & \qquad \texttt{self.c}{:=}\,\texttt{newC}\rangle;
\end{array}
$$

Figure 1: Nested abstract datatypes for `Point` and `ColorPoint` classes.

$$
\begin{array}{l}
\texttt{let addMove} = \Lambda().\,\lambda\texttt{ob}.\langle \texttt{ob} \longleftarrow\!\!+\ \texttt{move} = \Lambda().\,\lambda\texttt{s}.\,\lambda\texttt{dx}.\,\texttt{s} \Leftarrow \texttt{setX}(\texttt{dx} + \texttt{s} \Leftarrow \texttt{getX})\rangle \\
\quad \texttt{in let y} = \langle\, \texttt{x} = 3, \\
\qquad\qquad \texttt{getX} = \Lambda().\,\lambda\texttt{s}.\,\texttt{s.x} \\
\qquad\qquad \texttt{setX} = \Lambda().\,\lambda\texttt{s}.\,\lambda\texttt{nx}.\langle \texttt{s.x} := \texttt{nx}\rangle \\
\qquad \texttt{in addMove} * ()\, \texttt{y}
\end{array}
$$

$$
\begin{array}{rcl}
\texttt{addMove} & : & \forall (r <:_w \langle\!\langle \texttt{getX} : int,\ \texttt{setX} : int \rightarrow unit \rangle\!\rangle :: L)\textbf{pro.}\,r \rightarrow \textbf{pro.}\langle\!\langle r \mid \texttt{move} : int \rightarrow unit \rangle\!\rangle \\
\texttt{y} & : & \textbf{pro.}\langle\!\langle \texttt{x} : int,\ \texttt{getX} : int,\ \texttt{setX} : int \rightarrow unit \rangle\!\rangle \\
\texttt{addMove} * ()\,\texttt{y} & : & \textbf{pro.}\langle\!\langle \langle\!\langle \texttt{x} : int,\ \texttt{getX} : int,\ \texttt{setX} : int \rightarrow unit \rangle\!\rangle \mid \texttt{move} : int \rightarrow unit \rangle\!\rangle
\end{array}
$$

Figure 2: Mixin function `addMove` and the types of some of its subexpressions.

$int \rightarrow unit$ `setX` methods. Figure 2 also includes the types of various fragments of function `addMove`. In the example, L is any kind not containing the label `move`, as we are tracking components that may have already been added. (If we were tracking method absences, then the kind should *include* `move`.)

5 Operational Semantics

The operational semantics, presented in full in Appendix B, is inspired by the semantics for the imperative first-order calculus of [AC96a]. It is based solely on reduction relations, without any equational theory. The semantics relates expressions of the calculus to addresses of a global store in which the results of the evaluation are recorded. The Abadi-Cardelli calculus has only object terms and hence only one sort of result, which consists of sequences of store locations, one for each component of the object. Since our calculus is an extension of lambda calculus, we have values that are not objects (for instance, lambda abstractions and polymorphic functions). For the sake of uniformity, we have chosen to evaluate all terms to store addresses. These addresses then contain the results. In particular, the evaluation of an object term does not return a sequence of store locations but the address in which such a sequence of locations is stored. In the sequel, we will use the terms "location" and "address" synonymously. As in [AC96a], we avoid using formal substitution during reduction, keeping to standard implementation techniques for imperative languages.

In more detail, the semantics is based on *environments* and *stores*. An environment E associates term variables to addresses. A store S associates addresses to values. Values are the results to which expressions of the calculus evaluate (for instance, a lambda abstraction evaluates to a closure, an object to a sequence of labeled addresses, and so on). The operational semantics is expressed via reduction judgments of the form $\{\!|S;\, E|\!\} \models e \longrightarrow a \bullet S'$, indicating that expression e evaluates to address a starting from store S and environment E. In the process, store S is modified to produce store S'. The value associated with expression e is stored in location a of store S'.

For the most part, the reduction rules are intuitive. Rules (*lambda*) and ($\forall$ *intro*) evaluate term and row variable abstractions, respectively. Both rules store the appropriate closure in a fresh store location. Rule ($\leftarrow\!\!\!\ominus$), which appears below, describes how to evaluate method and field extensions and method override. Before discussing the rule, however, we need to introduce some notation. We use the symbol $\leftarrow\!\!\!\ominus$ as a meta-notation standing for any of $\longleftarrow\!\!\!\!+$, $\leftarrow$, or $\longleftarrow\!\!\!\!+\!:$. Symbol η denotes a finite map from addresses to values; we use η to denote object values. The notation $\eta[l = a]$ denotes the finite map just like η except that $\eta[l = a]$ maps l to a. With this notation, we may now consider rule ($\leftarrow\!\!\!\ominus$):

$$\frac{\begin{array}{c} \{\!|S;\, E|\!\} \models e_1 \longrightarrow a_1 \bullet S_1 \\ \{\!|S_1;\, E|\!\} \models e_2 \longrightarrow a_2 \bullet S_2 \\ S_2(a_1) = \eta \qquad\qquad a_3 \notin dom(S_2) \end{array}}{\{\!|S;\, E|\!\} \models \langle e_1 \leftarrow\!\!\!\ominus\, l = e_2 \rangle \longrightarrow a_3 \bullet (S_2, a_3 \mapsto \eta[l = a_2])}$$

Intuitively, we require e_1 to evaluate to an address containing an object (condition $S_2(a_1) = \eta$). Expression e_2 need only evaluate to an address. The type system will insure that $l \notin dom(\eta)$ if $\leftarrow\!\!\!\ominus$ is either $\longleftarrow\!\!\!\!+$ or $\longleftarrow\!\!\!\!+\!:$ and that $l \in dom(\eta)$ if $\leftarrow\!\!\!\ominus$ is $\leftarrow$. It will also insure that $S_2(a_1)$ is an object value, not a closure. The result of the evaluation is then a modification of a "clone" of the targeted object, allocated in a fresh address (a_3). This cloning is necessary for type soundness of the extension operations.

It is worth remarking that the field override operation (for which we have a different rule ($:=$)) changes the value of the field of the object to which it is applied, instead of creating and modifying a clone. Hence field override is inherently imperative.

The rule for method invocation ($\Leftarrow$) is slightly more complicated:

$$\frac{\begin{array}{c} \{\!|S;\, E|\!\} \models e \longrightarrow a_1 \bullet S_1 \\ S_1(a_1) = \eta \qquad\qquad\qquad \eta(m) = a_1' \\ S_1(a_1') = \langle \Lambda().\, e_1,\, E_1 \rangle \\ \{\!|S_1;\, E_1|\!\} \models e_1 \longrightarrow a_2 \bullet S_2 \\ S_2(a_2) = \langle \lambda x.\, e_2,\, E_2 \rangle \qquad\quad x \notin dom(E_2) \\ \{\!|S_2;\, E_2, x \mapsto a_1|\!\} \models e_2 \longrightarrow a_3 \bullet S_3 \end{array}}{\{\!|S;\, E|\!\} \models e \Leftarrow m \longrightarrow a_3 \bullet S_3}$$

First, we evaluate the receiver object to its address a_1 to obtain η, the list of methods and fields associated with object e. We then look up the row closure associated with method m in η and evaluate its body e_1 in the appropriate environment to find the term closure that stores the method body $\lambda x.\, e_2$. We then evaluate e_2 with x bound to the address of the receiver object in the appropriate environment. This binding is where the first parameter to a method body is bound to its host object, modeling *self*. The field selection evaluation rule (*field selection*) is simpler, since field bodies are not polymorphic.

6 Subject Reduction

To show that the operational semantics is consistent with the type system, we have to give types to results, *i.e.*, give a type to each store location. As noted in Chapter 11 of [AC96b], it is impossible to state the type of a result simply by recursively examining its subcomponents because stores may contain loops. To address this problem, we follow Abadi and Cardelli in introducing *store typings* to allow us to type results independently of a particular store. This approach is reasonable because type-safe computations store results of different types in different locations. Intuitively, all store locations are typed "globally" with respect to the whole store. Store types and the machinery to prove subject reduction is complicated in our case by the presence of types with free row variables. Since it only makes sense to give store locations closed types, we need:

- a universal quantifier to close method body types;
- a *row enviroment substitution*, to enable comparisons between static types, which can contain free row variables, and run-time store types (Appendix C.2), which cannot. This technique was inspired by the substitution operation presented in Chapter 17 of [AC96b] for their imperative calculus with *Self* type.

Using these tools, we may show that our system enjoys a subject reduction property. Informally, we may state it as follows:

> If we can statically give expression e type τ and e evaluates in store S with store type Σ to a result stored in address a of the modified store S', then there exists a type τ' that subtypes (a

closed version of) type τ and a store type Σ' such that Σ' extends Σ, correctly types S', and gives address a type τ'.

The formal statement of subject reduction is Theorem D.6 in Appendix D.

7 Related Work

In the literature, there are various approaches to modeling classes other than ours and the record-of-premethods approach that we briefly discussed in Section 1.1:

- *Existential Model*: In [PT94], Pierce and Turner interpret classes as object-generating functions. In their interpretation, inheritance is interpreted as modifications to the object-generating functions that model classes. This encoding is somewhat cumbersome, since it requires programmers to explicitly manipulate `get` and `put` functions, which intuitively convert between the hidden state of parent class objects and derived class objects. One solution to this problem is proposed in [HP96]: Hofmann and Pierce introduce a refined version of $F_{<:}$ that permits only positive subtyping. With this restriction, `get` and `put` functions are both guaranteed to exist and hence may be handled in a more automatic fashion in class encodings.

- *Direct Models*: Kim Bruce has developed a family of type-safe formal languages that model classes directly instead of via an interpretation as the combination of more basic primitives. In [Bru93], Bruce describes TOOPL, a functional object-oriented language. PolyTOIL, presented in [BSv95], incorporates imperative features and introduces the notion of *matching*, a relationship between object types that holds whenever the first is an extension of the second, regardless of the variance of the "mytype" type variable. In these languages, the type of an object reflects only its public interface; it cannot convey implementation information. Scott Smith and the Hopkins Object Group have designed a type-safe class-based object-oriented language with a rich feature set called I-Loop, [EST95]. Their type system is based on polymorphic recursively constrained types, for which they have a sound type inferencing algorithm. The main advantage of this approach is the extreme flexibility afforded by recursively constrained types. Currently, the main problem is that it returns large, difficult-to-read types. Some form of simplification may be required. Work in this area is in progress.

Our system falls in the category of prototype calculi with subtyping. We briefly mention here three other approaches to extensible objects compatible with some form of subtyping.

- In [BL95] a type system is presented for the functional Lambda Calculus of Objects [FHM94], in which a limited form of width subtyping compatible with object extension is introduced. Only methods that are not invoked by other methods in the object can be "forgotten" by subtyping. The machinery to track information about used methods is somewhat heavy, but it avoids constraints on the variance of the *self*-type, which is instead one of the weak points of [FM95a].

- The type system of [AC96b] is extended in [Liq96] to allow a form of extension in the presence of a limited form of width subtyping. This approach combines a variant of [BL95] with that of [FM95a]: an object may be in one of two states, one allowing extensions and limited width subtyping, the other forbidding extensions but supporting unrestricted width subtyping.

- PolyTOIL combines a form of (match-bounded) universal quantification and extensible structures. It uses the matching relation on types to type check inheritance of extensible classes and uses match-bound quantification to type check polymorphic functions.

8 Conclusion

We have presented a formal study of a calculus to support the "Classes = Extensible Objects + Encapsulation" interpretation of classes. Despite its apparent complexity (although this impression may be caused more by the number of rules rather than the complexity of the rules themselves), the calculus is rather intuitive, since:

- it uses well-established concepts such as universal and existential quantifiers,

- it enables straightforward translations of mixins and Java-like classes, and

- it has a clear imperative semantics and a well-developed formal meta-theory.

We have not completed our study of the system. Future work includes:

- adopting a module construct and a dot notation as in [CL90, HL94] to replace the present nested structure of class hierarchies,

- introducing a form of alpha-conversion to allow us to hide the private names that are currently exposed by our positive kinds, as in [RS98];

- proving a type soundness theorem, and

- developing a typing algorithm along the lines of the one in Compagnoni's thesis [Com94]. We conjecture our system is decidable [1].

References

[AC96a] M. Abadi and L. Cardelli. An imperative object calculus. *Theory and Practice of Object Systems*, 1(3):151–166, 1996. Earlier version appeared in TAPSOFT '95 proceedings.

[AC96b] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, 1996.

[AC96c] M. Abadi and L. Cardelli. A theory of primitive objects: Untyped and first-order systems. *Information and Computation*, 125(2):78–102, 1996. Earlier version appeared in TACS '94 proceedings, LNCS 789.

[AG96] K. Arnold and J. Gosling. *The Java Programming Language*. Addison Wesley, 1996.

[BBL96] V. Bono, M. Bugliesi, and L. Liquori. A lambda calculus of incomplete objects. In *Proc. MFCS '96*, pages 218–229. Springer LNCS 1113, 1996.

[1] As of the preparation of this version of the paper, we have completed the last two items listed under future work.

[BDMN73] G.M. Birtwistle, O.-J. Dahl, B. Myhrhaug, and K. Nygaard. *Simula Begin.* Studentlitteratur, Box 1717, S-222 01 Lund, Sweden; Auerbach, Philadelphia, 1973.

[BL95] V. Bono and L. Liquori. A subtyping for the Fisher-Honsell-Mitchell lambda calculus of objects. In L. Pacholsky and J. Tiuryn, editors, *Proc. of Int'l Conf. of Computer Science Logic*, pages 16–30, Berlin, June 1995. Springer LNCS 933.

[Bru93] K. Bruce. Safe type checking in a statically-typed object-oriented programming language. In *Proc. 20th ACM Symp. Principles of Programming Languages*, pages 285–298, 1993.

[BSv95] K. Bruce, A. Schuett, and R. van Gent. PolyTOIL: A type-safe polymorphic object-oriented language. In *Proc. 9th European Conference on Object-Oriented Programming*, pages 26–51, Aarhus, Denmark, 1995. Springer LNCS 952.

[CL90] L. Cardelli and X. Leroy. *Abstract types and the dot notation*, pages 479–504. IFIP State of the Art Reports. North Holland, March 1990. Also appeared as SRC Research Report 56.

[Com94] A.B. Compagnoni. *Higher-Order Subtyping with Intersection Types.* PhD thesis, Katholieke Universiteit Nijmegen, 1994.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[ES90] M. Ellis and B. Stroustrup. *The Annotated C^{++} Reference Manual.* Addison-Wesley, 1990.

[EST95] J. Eifrig, S. Smith, and V. Trifonov. Sound polymorphic type inference for objects. In *ACM Conf. Object-Oriented Programming: Systems, Languages and Applications*, pages 169–184, October 1995.

[FHM94] K. Fisher, F. Honsell, and J.C. Mitchell. A lambda calculus of objects and method specialization. *Nordic J. Computing (*formerly *BIT)*, 1:3–37, 1994. Preliminary version appeared in *Proc. IEEE Symp. on Logic in Computer Science,* 1993, 26–38.

[Fis96] K. Fisher. *Type Systems for object-oriented programming languages.* PhD thesis, Stanford University, 1996.

[FM95a] K. Fisher and J.C. Mitchell. A delegation-based object calculus with subtyping. In *Proc. 10th Int'l Conf. Fundamentals of Computation Theory (FCT'95)*, pages 42–61. Springer LNCS 965, 1995.

[FM95b] K. Fisher and J.C. Mitchell. The development of type systems for object-oriented languages. *Theory and Practice of Object Systems*, 1(3):189–220, 1995. Preliminary version appeared in TACS '94 proceedings.

[FM98] K. Fisher and J.C. Mitchell. On the relationship between classes, objects, and data abstraction. *Theory and Practice of Object Systems*, 1998. To Appear. Preliminary version appeared in Marktoberdorf '97 proceedings.

[HL94] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proc. 21st ACM Symp. on Principles of Programming Languages*, 1994.

[HP96] M. Hofmann and B.C. Pierce. Positive subtyping. *Information and Computation*, 126(1):11–33, 1996. Preliminary version appeared in *Proc. 22nd ACM Symp. on Principles of Programming Languages*, 1995.

[KLM94] D. Katiyar, D. Luckham, and J.C. Mitchell. A type system for prototyping languages. In *Proc. 21st ACM Symp. on Principles of Programming Languages*, 1994.

[Liq96] L. Liquori. An extended theory of primitive objects: First and second order systems. Technical Report CS-23-96, Dipartimento di Informatica, Universitá di Torino, 1996. A portion of this work appears in ECOOP '97 Proceedings, LNCS 1241.

[Mey92] B. Meyer. *Eiffel: The Language.* Prentice-Hall, 1992.

[PT93] B.C. Pierce and D.N. Turner. Statically typed friendly functions via partially abstract types. Technical Report ECS-LFCS-93-256, University of Edinburgh, LFCS, April 1993. Also available as INRIA-Rocquencourt Rapport de Recherche No. 1899.

[PT94] B.C. Pierce and D.N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–248, 1994. Preliminary version appeared in *Proc. 20th ACM Symp. on Principles of Programming Languages*, 1993, under the title "Object-oriented programming without recursive types".

[RR96] J.H. Reppy and J.G. Riecke. Classes in Object ML via modules, 1996. Presented at FOOL3 workshop.

[RS98] J.G. Riecke and C. Stone. Privacy via subsumption. In *Proc. 5th Annual FOOL Workshop*, January 1998. To appear.

[Str94] B. Stroustrup. *The Design and Evolution of C^{++}.* Addison-Wesley, 1994. Chapter 3: The birth of C^{++}.

[Wan78] M. Wand. A new incompleteness result for Hoare's system. *J. ACM*, 25(1):168–175, Jan. 1978.

A Typing rules

A.1 Context Rules

(start)
$$\frac{}{\epsilon \vdash *}$$

(row var)
$$\frac{\rho \vdash * \qquad r \notin dom(\rho) \qquad \rho \vdash R :: L_1 \qquad L_1 \subseteq L_0}{\rho\ ,\ (r <:_w R :: L_0) \vdash *}$$

(exp var)
$$\frac{\rho\ ;\ \Gamma \vdash * \qquad \rho \vdash \tau : T \qquad x \notin dom(\Gamma)}{\rho\ ;\ \Gamma,\ x : \tau \vdash *}$$

(empty exp)
$$\frac{\rho \vdash *}{\rho\ ;\ \epsilon \vdash *}$$

A.2 Rules for Type Expressions

(type unit)
$$\frac{\rho \vdash *}{\rho \vdash unit : T}$$

(type arrow)
$$\frac{\rho \vdash \tau_1 : T \qquad \rho \vdash \tau_2 : T}{\rho \vdash \tau_1 \rightarrow \tau_2 : T}$$

(probj)
$$\frac{\rho \vdash R :: L}{\rho \vdash \mathbf{probj}.R : T}$$

$$(\forall)\qquad \frac{\rho\,,\,(r <:_w R :: L) \vdash \tau : T}{\rho \vdash \forall(r <:_w R :: L)\tau : T}$$

$$(\exists)\qquad \frac{\rho\,,\,(r <:_w R :: L) \vdash \tau : T}{\rho \vdash \exists(r <:_w R :: L)\tau : T}$$

A.3 Rules for Row Expressions

$$(\mathit{row\ proj})\qquad \frac{\rho \vdash * \qquad r <:_w R :: L \in \rho}{\rho \vdash r :: L}$$

$$(\mathit{empty\ row})\qquad \frac{\rho \vdash *}{\rho \vdash \langle\!\langle\rangle\!\rangle :: \emptyset}$$

$$(\mathit{row\ label})\qquad \frac{\rho \vdash R :: L_1 \qquad L_1 \subseteq L_2}{\rho \vdash R :: L_2}$$

$$(\mathit{row\ ext})\qquad \frac{\rho \vdash R :: L \qquad l \notin L \qquad \rho \vdash \tau : T}{\rho \vdash \langle\!\langle R \mid l : \tau\rangle\!\rangle :: L \cup \{l\}}$$

A.4 Subtyping Rules for Rows and Types

$$(<:\ \mathit{refl})\qquad \frac{\rho \vdash U : -\gamma}{\rho \vdash U <:_{(B)} U}$$

$$(<:\ \mathit{trans})\qquad \frac{\rho \vdash U_1 <:_{(B)} U_2 \qquad \rho \vdash U_2 <:_{(B')} U_3}{\rho \vdash U_1 <:_{(B+B')} U_3}$$

A.5 Subtyping Rules for Types

$$(<:\rightarrow)\qquad \frac{\rho \vdash \tau_1' <: \tau_1 \qquad \rho \vdash \tau_2 <: \tau_2'}{\rho \vdash \tau_1 \rightarrow \tau_2 <: \tau_1' \rightarrow \tau_2'}$$

$$(<:\ \mathit{obj})\qquad \frac{\rho \vdash R_1 <:_B R_2}{\rho \vdash \mathbf{probj.}R_1 <: \mathbf{obj.}R_2}$$

$$(<:\ \mathit{unit})\qquad \frac{\rho \vdash \tau : T}{\rho \vdash \tau <: \mathit{unit}}$$

A.6 Subtyping Rules for Rows

$$(\mathit{row\ proj\ bound})\qquad \frac{\rho \vdash * \qquad r <:_w R :: L \in \rho}{\rho \vdash r <:_w R}$$

$$(<:\ \mathit{cong})\qquad \frac{\rho \vdash R_1 <:_B R_2 \qquad \rho \vdash \langle\!\langle R_i \mid l : \tau\rangle\!\rangle :: L_i \quad i \in \{1,2\}}{\rho \vdash \langle\!\langle R_1 \mid l : \tau\rangle\!\rangle <:_B \langle\!\langle R_2 \mid l : \tau\rangle\!\rangle}$$

$$(<:\ d)\qquad \frac{\rho \vdash R_1 <:_B R_2 \qquad \rho \vdash \tau_1 <: \tau_2 \qquad \rho \vdash \langle\!\langle R_i \mid m : \tau_i\rangle\!\rangle :: L_i \quad i \in \{1,2\}}{\rho \vdash \langle\!\langle R_1 \mid m : \tau_1\rangle\!\rangle <:_{B+d} \langle\!\langle R_2 \mid m : \tau_2\rangle\!\rangle}$$

$$(<:\ w)\qquad \frac{\rho \vdash R_1 <:_B R_2 \qquad \rho \vdash \langle\!\langle R_1 \mid l : \tau\rangle\!\rangle :: L_1}{\rho \vdash \langle\!\langle R_1 \mid l : \tau\rangle\!\rangle <:_B R_2}$$

A.7 Rules for Assigning Types to Terms

$$(\mathit{exp\ proj})\qquad \frac{\rho\,;\,\Gamma \vdash * \qquad x : \tau \in \Gamma}{\rho\,;\,\Gamma \vdash x : \tau}$$

$$(\mathit{exp\ abs})\qquad \frac{\rho\,;\,\Gamma,\ x : \tau_1 \vdash e : \tau_2}{\rho\,;\,\Gamma \vdash \lambda x.\,e : \tau_1 \rightarrow \tau_2}$$

$$(\mathit{exp\ app})\qquad \frac{\rho\,;\,\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \qquad \rho\,;\,\Gamma \vdash e_2 : \tau_1}{\rho\,;\,\Gamma \vdash e_1\, e_2 : \tau_2}$$

$$(\mathit{let})\qquad \frac{\rho\,;\,\Gamma \vdash e_1 : \tau_1 \qquad \rho\,;\,\Gamma,\ x : \tau_1 \vdash e_2 : \tau_2}{\rho\,;\,\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau_2}$$

$$(\mathit{empty\ pro})\qquad \frac{\rho\,;\,\Gamma \vdash *}{\rho\,;\,\Gamma \vdash \langle\rangle : \mathbf{pro.}\langle\!\langle\rangle\!\rangle}$$

$$(\mathit{meth\ ext})\qquad \frac{\rho\,;\,\Gamma \vdash e_1 : \mathbf{pro.}R \qquad \rho \vdash R :: L \qquad m \notin L \qquad \rho\,;\,\Gamma \vdash e_2 : \forall(r <:_w \langle\!\langle R \mid m : \tau\rangle\!\rangle :: \mathit{All})\ \mathbf{pro.}r \rightarrow \tau}{\rho\,;\,\Gamma \vdash \langle e_1 \longleftrightarrow m = e_2\rangle : \mathbf{pro.}\langle\!\langle R \mid m : \tau\rangle\!\rangle}$$

$$(\mathit{meth\ over})\qquad \frac{\rho\,;\,\Gamma \vdash e_1 : \mathbf{pro.}R \qquad \rho \vdash R <:_w \langle\!\langle m : \tau\rangle\!\rangle \qquad \rho\,;\,\Gamma \vdash e_2 : \forall(r <:_w R :: \mathit{All})\ \mathbf{pro.}r \rightarrow \tau}{\rho\,;\,\Gamma \vdash \langle e_1 \leftarrow m{=}e_2\rangle : \mathbf{pro.}R}$$

$$\textit{(field ext)}\ \frac{\rho\ ;\Gamma\vdash e_1 : \mathbf{pro.}R \quad \rho\vdash R::L \quad f\notin L \quad \rho\ ;\Gamma\vdash e_2:\tau}{\rho\ ;\Gamma\vdash\langle e_1 \longleftrightarrow\!\!+\!: f=e_2\rangle : \mathbf{pro.}\langle\!\langle R\,|\,f:\tau\rangle\!\rangle}$$

$$\textit{(field over)}\ \frac{\rho\ ;\Gamma\vdash e_1 : \mathbf{probj.}R \quad \rho\vdash R <:_w \langle\!\langle f:\tau\rangle\!\rangle \quad \rho\ ;\Gamma\vdash e_2:\tau}{\rho\ ;\Gamma\vdash\langle e_1.f:=e_2\rangle : \mathbf{probj.}R}$$

$$(\Leftarrow\ \textit{select})\ \frac{\rho\ ;\Gamma\vdash e : \mathbf{probj.}R \quad \rho\vdash R <:_w \langle\!\langle m:\tau\rangle\!\rangle}{\rho\ ;\Gamma\vdash e\Leftarrow m : \tau}$$

$$\textit{(field select)}\ \frac{\rho\ ;\Gamma\vdash e : \mathbf{probj.}R \quad \rho\vdash R <:_w \langle\!\langle f:\tau\rangle\!\rangle}{\rho\ ;\Gamma\vdash e.f : \tau}$$

$$(\forall\ \textit{intro})\ \frac{\rho\,,(r <:_w R::L)\,;\Gamma\vdash e:\tau \quad r\notin FV(\Gamma)}{\rho\ ;\Gamma\vdash\Lambda().\,e : \forall(r <:_w R::L)\tau}$$

$$(\forall\ \textit{elim})\ \frac{\rho\ ;\Gamma\vdash e : \forall(r <:_w R::L)\tau \quad \rho\vdash R' <:_w R \quad \rho\vdash R'::L}{\rho\ ;\Gamma\vdash e*() : [R'/r]\tau}$$

$$(\exists\ \textit{intro})\ \frac{\rho\vdash R' <:_w R \quad \rho\vdash R'::L \quad \rho\ ;\Gamma\vdash e : [R'/r]\tau}{\rho\ ;\Gamma\vdash \{\!|(r <:_w R::L)=R',\ e|\!\} : \exists(r <:_w R::L)\tau}$$

$$(\exists\ \textit{elim})\ \frac{\rho\ ;\Gamma\vdash e_1 : \exists(r <:_w R::L)\tau \quad \rho\,,(r <:_w R::L)\,;\Gamma,\,x:\tau\vdash e_2:\tau_2 \quad \rho\vdash\tau_2 : T}{\rho\ ;\Gamma\vdash \mathbf{Abstype}(r <:_w R::L)\ \mathbf{with}\ x:\tau\ \mathbf{is}\ e_1\ \mathbf{in}\ e_2 : \tau_2}$$

$$\textit{(subsumption)}\ \frac{\rho\ ;\Gamma\vdash e:\tau_1 \quad \rho\vdash\tau_1 <: \tau_2}{\rho\ ;\Gamma\vdash e:\tau_2}$$

B Operational Semantics

Addresses	a	$::=$	an integer, selected from one of four disjoint sets: S_{OBJ}, S_{RCL}, S_{CL}, or S_{EXIS}
Storable Values	sv	$::=$	$\eta\,|\,c\,|\,rc\,|\,a$
Object Value	η	$::=$	$[l_i\ =\ a_i]^{i\in I}$
Closure Value	c	$::=$	$\langle\lambda x.\,e,\,E\rangle$
Row Closure Value	rc	$::=$	$\langle\Lambda().\,e,\,E\rangle$
Environment	E	$::=$	$x_i\mapsto a_i,\ i\in 1,\ldots,n$
Store	S	$::=$	$a_i\mapsto sv_i,\ i\in 1,\ldots,n$
Contexts	$\{\!|S;\,E|\!\}$	$::=$	store/environment pair.

B.1 Semantic Judgments

$S\models *$	well-formed store
$\{\!|S;\,E|\!\}\models *$	well-formed store/env pair.
$\{\!|S;\,E|\!\}\models e\longrightarrow a\bullet S'$	expression e evaluates to address a and produces store S'.

B.2 Context Judgments

$$\textit{(empty store)}\ \frac{}{\emptyset\models *}$$

$$\textit{(closure)}\ \frac{\{\!|S;\,E|\!\}\models * \quad a\notin dom(S)}{S,\,a\mapsto\langle\lambda x.\,e,\,E\rangle\models *}$$

$$(\Lambda\!-\!\textit{closure})\ \frac{\{\!|S;\,E|\!\}\models * \quad a\notin dom(S)}{S,\,a\mapsto\langle\Lambda().\,e,\,E\rangle\models *}$$

$$\textit{(object)}\ \frac{S\models * \quad a\notin dom(S) \quad a_i\in dom(S)\text{ for all }i\in I}{S,\,a\mapsto[l_i=a_i]^{i\in I}\models *}$$

$$\textit{(exist)}\ \frac{S\models * \quad a\notin dom(S) \quad a'\in dom(S)}{S,\,a\mapsto a'\models *}$$

$$\textit{(empty environment)}\ \frac{S\models *}{\{\!|S;\,\emptyset|\!\}\models *}$$

$$\textit{(var)}\ \frac{\{\!|S;\,E|\!\}\models * \quad x\notin dom(E) \quad a\in dom(S)}{\{\!|S;\,E,x\mapsto a|\!\}\models *}$$

B.3 Reduction Rules

$$(var)\quad \frac{\begin{array}{c}\{\!|S;\ E|\!\} \models * \\ E(x) = a\end{array}}{\{\!|S;\ E|\!\} \models x \longrightarrow a \bullet S}$$

$$(lambda)\quad \frac{\begin{array}{c}\{\!|S;\ E|\!\} \models * \\ a \notin dom(S)\end{array}}{\{\!|S;\ E|\!\} \models \lambda x.\, e \longrightarrow a \bullet (S, a \mapsto \langle \lambda x.\, e,\ E\rangle)}$$

$$(app)\quad \frac{\begin{array}{c}\{\!|S;\ E|\!\} \models e_1 \longrightarrow a_1 \bullet S_1 \\ \{\!|S_1;\ E|\!\} \models e_2 \longrightarrow a_2 \bullet S_2 \\ S_2(a_1) = \langle \lambda x.\, e,\ E'\rangle \qquad x \notin dom(E') \\ \{\!|S_2;\ E',\ x \mapsto a_2|\!\} \models e \longrightarrow a_3 \bullet S_3\end{array}}{\{\!|S;\ E|\!\} \models e_1\ e_2 \longrightarrow a_3 \bullet S_3}$$

$$(empty\ object)\quad \frac{\begin{array}{c}\{\!|S;\ E|\!\} \models * \\ a \notin dom(S)\end{array}}{\{\!|S;\ E|\!\} \models \langle\rangle \longrightarrow a \bullet (S, a \mapsto [\,])}$$

In the following rule, we use symbol $\leftarrow\!\!\circ$ as a meta-notation standing for any of $\longleftarrow\!\!\!+$, $\leftarrow$, or $\longleftarrow\!\!\!+\!:$. The notation $\eta[l = a]$ denotes the finite map just like η except that $\eta[l = a]$ maps l to a.

$$(\leftarrow\!\!\circ)\quad \frac{\begin{array}{c}\{\!|S;\ E|\!\} \models e_1 \longrightarrow a_1 \bullet S_1 \\ \{\!|S_1;\ E|\!\} \models e_2 \longrightarrow a_2 \bullet S_2 \\ S_2(a_1) = \eta \qquad a_3 \notin dom(S_2)\end{array}}{\{\!|S;\ E|\!\} \models \langle e_1 \leftarrow\!\!\circ\ l = e_2\rangle \longrightarrow a_3 \bullet (S_2, a_3 \mapsto \eta[l = a_2])}$$

In the following rule, the notation $S_2[a_1 \mapsto \eta[f = a_2]]$ denotes the finite map just like S_2 except that a_1 is mapped now to $\eta[f = a_2]$.

$$(:=)\quad \frac{\begin{array}{c}\{\!|S;\ E|\!\} \models e_1 \longrightarrow a_1 \bullet S_1 \\ \{\!|S_1;\ E|\!\} \models e_2 \longrightarrow a_2 \bullet S_2 \\ S_2(a_1) = \eta\end{array}}{\{\!|S;\ E|\!\} \models \langle e_1.f := e_2\rangle \longrightarrow a_1 \bullet S_2[a_1 \mapsto \eta[f = a_2]]}$$

$$(\Leftarrow)\quad \frac{\begin{array}{c}\{\!|S;\ E|\!\} \models e \longrightarrow a_1 \bullet S_1 \\ S_1(a_1) = \eta \qquad \eta(m) = a_1' \\ S_1(a_1') = \langle \Lambda().\, e_1,\ E_1\rangle \\ \{\!|S_1;\ E_1|\!\} \models e_1 \longrightarrow a_2 \bullet S_2 \\ S_2(a_2) = \langle \lambda x.\, e_2,\ E_2\rangle \qquad x \notin dom(E_2) \\ \{\!|S_2;\ E_2, x \mapsto a_1|\!\} \models e_2 \longrightarrow a_3 \bullet S_3\end{array}}{\{\!|S;\ E|\!\} \models e \Leftarrow m \longrightarrow a_3 \bullet S_3}$$

$$(field\ selection)\quad \frac{\begin{array}{c}\{\!|S;\ E|\!\} \models e \longrightarrow a_1 \bullet S_1 \\ S_1(a_1) = \eta \qquad \eta(f) = a_2\end{array}}{\{\!|S;\ E|\!\} \models e.f \longrightarrow a_2 \bullet S_1}$$

$$(let)\quad \frac{\begin{array}{c}\{\!|S;\ E|\!\} \models e_1 \longrightarrow a_1 \bullet S_1 \\ \{\!|S;\ E,\ x \mapsto a_1|\!\} \models e_2 \longrightarrow a_2 \bullet S_2 \\ x \notin dom(E)\end{array}}{\{\!|S;\ E|\!\} \models \textbf{let}\ x\ =\ e_1\ \textbf{in}\ e_2 \longrightarrow a_2 \bullet S_2}$$

$$(\forall\ intro)\quad \frac{\begin{array}{c}\{\!|S;\ E|\!\} \models * \\ a \notin dom(S)\end{array}}{\{\!|S;\ E|\!\} \models \Lambda().\, e \longrightarrow a \bullet (S, a \mapsto \langle \Lambda().\, e,\ E\rangle)}$$

$$(\forall\ elim)\quad \frac{\begin{array}{c}\{\!|S;\ E|\!\} \models e_1 \longrightarrow a_1 \bullet S_1 \\ S_1(a_1) = \langle \Lambda().\, e,\ E'\rangle \\ \{\!|S_1;\ E'|\!\} \models e \longrightarrow a_2 \bullet S_2\end{array}}{\{\!|S;\ E|\!\} \models e_1 * () \longrightarrow a_2 \bullet S_2}$$

$$(\exists\ intro)\quad \frac{\begin{array}{c}\{\!|S;\ E|\!\} \models e \longrightarrow a_1 \bullet S_1 \\ a_2 \notin dom(S_1)\end{array}}{\begin{array}{l}\{\!|S;\ E|\!\} \models \{\!|(r <:_w R :: L) = R',\ e|\!\} \longrightarrow \\ \qquad\qquad a_2 \bullet (S_1, a_2 \mapsto a_1)\end{array}}$$

$$(\exists\ elim)\quad \frac{\begin{array}{c}\{\!|S;\ E|\!\} \models e_1 \longrightarrow a_1 \bullet S_1 \\ S_1(a_1) = a_1' \qquad x \notin dom(E) \\ \{\!|S_1;\ (E, x \mapsto a_1')|\!\} \models e_2 \longrightarrow a_2 \bullet S_2\end{array}}{\begin{array}{l}\{\!|S;\ E|\!\} \models \textbf{Abstype}(r <:_w R :: L) \\ \qquad \textbf{with}\ x : \tau\ \textbf{is}\ e_1\ \textbf{in}\ e_2 \longrightarrow a_2 \bullet S_2\end{array}}$$

C Semantic Typing

Store Types

Σ	$::=$	$a_i \mapsto \tau_i,\ i \in 1, \ldots, n$ τ_i closed. Also, if τ_i is of the form $\exists(r <:_w R :: L)\tau$, then there is a subsidiary map $\Sigma_\exists$:
$\Sigma_\exists$	$::=$	$a_i \mapsto R_i$ R_i closed. Intuitively, R_i is the hidden implementation row associated with existential type $\exists(r <:_w R :: L)\tau$.

Row Environment

E_ρ	$::=$	$r_i \mapsto R_i,\ i \in 1, \ldots, m$ R_i closed.

C.1 Semantic Typing Judgments

$\Sigma \models *$	If $\forall a \in dom(\Sigma)$, $\epsilon \vdash \Sigma(a) : T$, and if $\Sigma(a) = \exists(r <:_w R :: L)\tau$, then $\epsilon \vdash \Sigma_\exists(a) <:_w R$ and $\epsilon \vdash \Sigma_\exists(a) :: L$.
$\Sigma' \succeq \Sigma$	If $dom(\Sigma) \subseteq dom(\Sigma')$, $\forall a \in dom(\Sigma), \Sigma(a) = \Sigma'(a)$, and $\forall a \in dom(\Sigma_\exists)$, $\Sigma_\exists(a) = \Sigma'_\exists(a)$.
$\Sigma \models E_\rho \bullet E : \rho \bullet \Gamma$	Row environment E_ρ and value environment E consistent with row context ρ and expression context Γ.
$\Sigma \models_{ob} S$	Object portion of S well-typed with respect to store type Σ.
$\Sigma \models_{cl} S$	Closure portion of S well-typed with respect to store type Σ.
$\Sigma \models_{rcl} S$	Row closure portion of S well-typed with respect to store type Σ.
$\Sigma \models_\exists S$	Existential portion of S well-typed with respect to store type Σ.
$\Sigma \models S$	S well-typed with respect to store type Σ.

C.2 Row Environment Substitution

$[E_\rho \rightarrow]$ is defined by:

$$
\begin{array}{ll}
[E_\rho \rightarrow]* & \stackrel{def}{=} * \\
[E_\rho \rightarrow]\tau & \stackrel{def}{=} [R_i/r_i]^{i \in 1..n}\,\tau \\
 & \quad \text{for } E_\rho \equiv r_i \mapsto R_i, i \in 1..n \\
[E_\rho \rightarrow]R & \stackrel{def}{=} [R_i/r_i]^{i \in 1..n}\,R \\
 & \quad \text{for } E_\rho \equiv r_i \mapsto R_i, i \in 1..n \\
[E_\rho \rightarrow]e & \stackrel{def}{=} [R_i/r_i]^{i \in 1..n}\,e \\
 & \quad \text{for } E_\rho \equiv r_i \mapsto R_i, i \in 1..n \\
[E_\rho \rightarrow](\tau_1 <: \tau_2) & \stackrel{def}{=} [E_\rho \rightarrow]\tau_1 <: [E_\rho \rightarrow]\tau_2 \\
[E_\rho \rightarrow](R_1 <:_B R_2) & \stackrel{def}{=} [E_\rho \rightarrow]R_1 <:_B [E_\rho \rightarrow]R_2 \\
[E_\rho \rightarrow](e : \tau) & \stackrel{def}{=} [E_\rho \rightarrow]e : [E_\rho \rightarrow]\tau \\
[E_\rho \rightarrow]\epsilon & \stackrel{def}{=} \epsilon \\
[E_\rho \rightarrow](\Gamma,\, x : \tau) & \stackrel{def}{=} [E_\rho \rightarrow]\Gamma,\, x : [E_\rho \rightarrow]\tau \\
[E_\rho \rightarrow]\epsilon & \stackrel{def}{=} \epsilon \\
[E_\rho \rightarrow](\rho,\, (r <:_w R :: L)) & \stackrel{def}{=} [E_\rho \rightarrow]\rho \\
 & \quad \text{if } r \in dom(E_\rho) \\
[E_\rho \rightarrow](\rho,\, (r <:_w R :: L)) & \stackrel{def}{=} [E_\rho \rightarrow]\rho,\, (r <:_w [E_\rho \rightarrow]R :: L) \\
 & \quad \text{if } r \notin dom(E_\rho) \\
[E_\rho \rightarrow](\rho\,;\Gamma\,) & \stackrel{def}{=} [E_\rho \rightarrow]\rho;\, [E_\rho \rightarrow]\Gamma
\end{array}
$$

C.3 Semantic Consistency Judgments

$$(Empty)\quad \frac{\Sigma \models *}{\Sigma \models \emptyset \bullet \emptyset : \epsilon \bullet \epsilon}$$

$$(Row\ Var)\quad \frac{\begin{array}{c}\Sigma \models E_\rho \bullet E : \rho \bullet \Gamma \\ \epsilon \vdash R_1 <:_w [E_\rho \rightarrow]R_2 \\ \epsilon \vdash R_1 :: L \qquad r \notin dom(\rho)\end{array}}{\begin{array}{c}\Sigma \models (E_\rho,\, r \mapsto R_1) \bullet E: \\ (\rho,\, (r <:_w R_2 :: L)) \bullet \Gamma\end{array}}$$

$$(Exp\ Var)\quad \frac{\begin{array}{c}\Sigma \models E_\rho \bullet E : \rho \bullet \Gamma \\ \epsilon \vdash \Sigma(a) <: [E_\rho \rightarrow]\tau \qquad x \notin dom(\Gamma)\end{array}}{\Sigma \models E_\rho \bullet (E, x \mapsto a) : \rho \bullet (\Gamma,\, x : \tau)}$$

C.4 Store Typing Judgments

$$(\models_{ob})\quad \frac{\begin{array}{c}\Sigma \models * \\ \Sigma(a_j) = \mathbf{pro.}\langle\!\langle m_i : \tau_i,\ f_k : \tau'_k \rangle\!\rangle^{i \in I,\, k \in K} \\ j \in 1..|S_{OBJ}| \\ \Sigma(a_i) = \forall(r_i <:_w R_i :: All)\mathbf{pro.}r_i \rightarrow \tau_i \quad i \in I \\ \epsilon \vdash \Sigma(a_k) <: \tau'_k \qquad k \in K \\ \epsilon \vdash \langle\!\langle m_i : \tau_i,\ f_k : \tau'_k \rangle\!\rangle^{i \in I,\, k \in K} <:_w R_i\end{array}}{\begin{array}{c}\Sigma \models_{ob} a_j \mapsto [m_i = a_i,\ f_k = a_k]^{i \in I,\, k \in K} \\ j \in 1..|S_{OBJ}|\end{array}}$$

$$(\models_{cl})\quad \frac{\begin{array}{c}\Sigma \models * \\ \Sigma(a_j) = \tau_j \rightarrow \tau'_j \qquad j \in 1..|S_{CL}| \\ \Sigma \models \emptyset \bullet E_j : \epsilon \bullet \Gamma_j \\ \epsilon;\, \Gamma_j,\, x_j : \tau_j \vdash e_j : \tau'_j\end{array}}{\begin{array}{c}\Sigma \models_{cl} a_j \mapsto \langle \lambda x_j.\, e_j,\, E_j \rangle \\ j \in 1..|S_{CL}|\end{array}}$$

$$(\models_{rcl})\quad \frac{\begin{array}{c}\Sigma \models * \\ \Sigma(a_j) = \forall(r_j <:_w R_j :: L_j)\tau_j \quad j \in 1..|S_{RCL}| \\ \Sigma \models \emptyset \bullet E_j : \epsilon \bullet \Gamma_j \\ (r_j <:_w R_j :: L_j)\,;\, \Gamma_j \vdash e_j : \tau_j\end{array}}{\begin{array}{c}\Sigma \models_{rcl} a_j \mapsto \langle \Lambda().\, e_j,\, E_j \rangle \\ j \in 1..|S_{RCL}|\end{array}}$$

$$(\models_{\exists})\quad \frac{\begin{array}{c}\Sigma \models * \\ \Sigma(a_j) = \exists(r_j <:_w R_j :: L_j)\tau_j \quad j \in 1..|S_{EXIS}| \\ \Sigma_\exists(a_j) = R'_j \\ \epsilon \vdash \Sigma(a'_j) <: [R'_j/r_j]\tau_j\end{array}}{\begin{array}{c}\Sigma \models_\exists a_j \mapsto a'_j \\ j \in 1..|S_{EXIS}|\end{array}}$$

$$(\models)\quad \frac{\begin{array}{cc}\Sigma \models_{ob} S & \Sigma \models_{cl} S \\ \Sigma \models_{rcl} S & \Sigma \models_\exists S\end{array}}{\Sigma \models S}$$

D Meta-theory

In this appendix, we briefly list some of the key lemmas needed to prove subject reduction and the subject reduction thereom itself.

Lemma D.1 (Bound Transformation) *If the judgments $\rho, (r <:_w R :: L), \rho'; \Gamma \vdash A$, $\rho \vdash R_1 <:_w R$, and $\rho \vdash R_1 :: L_1$ are all derivable, where $L_1 \subseteq L$, then we may derive the judgment $\rho, (r <:_w R_1 :: L), \rho'; \Gamma \vdash A$ as well.*

Lemma D.2 (Row Substitution) *If we may derive the judgments $\rho, (r <:_w R :: L), \rho'; \Gamma \vdash A$, $\rho \vdash R_1 <:_w R$, and $\rho \vdash R_1 :: L_1$, then we may derive $\rho, [R_1/r]\rho'; [R_1/r]\Gamma \vdash [R_1/r]A$ as well.*

Lemma D.3 (Strengthening) *If judgments $\rho; \Gamma, x : \tau, \Gamma' \vdash A$ and $\rho \vdash \tau' <: \tau$ are derivable, then so is $\rho; \Gamma, x : \tau', \Gamma' \vdash A$.*

Lemma D.4 (Properties of Consistency Judgments) *If judgment $\Sigma \models E_\rho \bullet E : \rho \bullet \Gamma$ is derivable, then*

1. *$r \in dom(E_\rho)$ iff $r \in dom(\rho)$,*
2. *$\Sigma \models \emptyset \bullet E : \epsilon \bullet [E_\rho \rightarrow]\Gamma$ is derivable as well,*
3. *if $\rho \vdash A$ is derivable, then so is $[E_\rho \rightarrow](\rho) \vdash [E_\rho \rightarrow]A$, and*
4. *if we may derive judgment $\rho\,; \Gamma \vdash A$, then we may also derive judgment $[E_\rho \rightarrow](\rho)\,; [E_\rho \rightarrow](\Gamma) \vdash [E_\rho \rightarrow]A$.*

Lemma D.5 (Store Type Weakening) *If we may derive the judgments $\Sigma \models E_\rho \bullet E : \rho \bullet \Gamma$ and $\Sigma' \models *$, and $\Sigma' \succeq \Sigma$ holds, then we may derive $\Sigma' \models E_\rho \bullet E : \rho \bullet \Gamma$ as well.*

Theorem D.6 (Subject Reduction) *If we may derive judgments $\rho\,; \Gamma \vdash e : \tau$, $\{S; E\} \models e \longrightarrow a \bullet S'$, $\Sigma \models E_\rho \bullet E : \rho \bullet \Gamma$, and $\Sigma \models S$ and $dom(\Sigma) = dom(S)$, then there exists a closed type τ' and a store type Σ' such that $\Sigma' \succeq \Sigma$, $dom(\Sigma') = dom(S')$, $\Sigma'(a) = \tau'$, and judgments $\Sigma' \models S'$ and $\epsilon \vdash \tau' <: [E_\rho \rightarrow]\tau$ are derivable.*

Proof The proof is by induction on the derivation of judgment $\{S; E\} \models e \longrightarrow a \bullet S'$.

Privacy via Subsumption

Jon G. Riecke
Bell Laboratories
Lucent Technologies
700 Mountain Avenue
Murray Hill, NJ 07974 USA
riecke@bell-labs.com

Christopher A. Stone
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213 USA
cstone@cs.cmu.edu

Abstract

We describe an object calculus that allows both extension of objects and full width subtyping (hiding arbitrary components). In contrast to other proposals, the types of our calculus do not mention "missing" methods. To avoid type unsoundness, the calculus mediates all interaction with objects via "dictionaries" that resemble the method dispatch tables in conventional implementations. Private fields and methods can be modeled and enforced by scoping restrictions: forgetting a field or method through subsumption makes it private. We prove that the type system is sound, give a variant which allows covariant self types, and give some examples of the expressiveness of the calculus.

1 Introduction

One of the most important principles of software engineering is information hiding: the ability to build and enforce data or procedural abstractions in order to make programs readable and maintainable. Most object-oriented programming languages provide direct support for information hiding. Class-based languages like C++ and Java [14], for instance, have mechanisms for hiding methods and fields via **private** annotations; private methods and fields added to an object may be accessed only by other methods defined within the same class.

In this paper we give an elementary account of private fields and methods in the presence of structural subtyping. We extend recent object calculi [1, 20] with operations for hiding and renaming methods, operations that can also be found in the class system of Eiffel [19]. The primitives of the calculus also include object extension, method override, and arbitrary width subtyping and subsumption (i.e., objects with more methods can always be used in contexts expecting fewer methods). We prove that the type systems prevent run-time type errors.

We take the somewhat novel approach of distinguishing between internal and external names of methods. In some sense, common implementations of object-oriented languages already use this idea: classes with methods are compiled into a table, with a pointer to each method, and the internal names are their location within the table. Our calculus distinguishes internal from external names so that methods can be made private, even after an object or class has been created. Once the external name of the method has been lost (either through an explicit operation or implicitly by subsumption), the method cannot be overridden; other methods referring to this method by its internal name are unaffected by the addition of new methods, whatever their external name.

The distinction between internal and external names allows us to avoid a well-known problem with object extension. In previous formalizations of extensible objects, the combination of object extension and unrestricted width subtyping was unsound. For instance, suppose we ignore this distinction and define an object p containing two methods, x which always returns 3, and $getx$ which returns the value of the x method:

$$\mathsf{obj}\, s.\{\!|x \triangleright 3 : \mathsf{Int}, getx \triangleright s.x : \mathsf{Int}|\!\}$$

The variable s stands for "self", and is dynamically bound to the object upon method invocation. The type of p is $\{\!|x : \mathsf{Int}, getx : \mathsf{Int}|\!\}$. If p can also have the less precise type $\{\!|getx : \mathsf{Int}|\!\}$, there is be nothing to prevent us from adding a new method x returning the value True of type Bool. In the dynamic semantics of [10, 18] where an object can have at most one x component at a time, this would override the earlier x method and cause $getx$ to thereafter return the value True; This is a type error as $getx$ is statically typed as returning an integer. To avoid such errors, the type systems of [10, 18] weaken the subtyping relation for extensible objects: either methods may not be hidden at all, or components can be made "inaccessible" (cannot be invoked) though still visible and overridable. These mechanisms do prevent two methods with the same name being added to the same object, but are unsatisfactory from a software-engineering standpoint: they require the implementor to expose implementation details in interfaces. Moreover, when we use these calculi as a basis for classes, additions or changes to the private methods of a base class may require subclasses to be re-typechecked and possibly recompiled (a problem familiar from C++ [16]). In the worst case, derived classes become ill-formed and large pieces of code must be rewritten.

We approach this problem by allowing unrestricted hiding and compensating in the dynamic semantics. When the above example has been translated into our system, the $getx$ method refers to x via an internal name so that $getx$ continues to return 3, even when the new x method is added. Because the semantics of object extension gives the new method a new internal name, the $getx$ method remains unaffected.

2 A Language of First-Order Extensible Objects

We begin with a first-order calculus (in the sense of [1]), i.e., the calculus without a notion of self type. For simplicity, we limit the

Presented at FOOL'5, 1998

Table 1: Syntax of First-Order System

$\tau ::=$		b	base type
	$\mid$	$(\tau \rightarrow \tau')$	function type
	$\mid$	$\{\vert l : \tau_l \,^{l \in I} \vert\}$	object type
$\Gamma ::=$		$\bullet$	typing contexts
	$\mid$	$\Gamma, x{:}\tau$	
$v ::=$		c	constant
	$\mid$	$x, y, s, o, \ldots$	variables
	$\mid$	$(\lambda x{:}\tau.e)$	abstraction
	$\mid$	$\mathsf{obj}\, s.\{\vert i \triangleright e_i : \tau_i \,^{i \in 1..n} \vert\}_\varphi$	object
$e ::=$		v	value
	$\mid$	$(e\, e')$	function application
	$\mid$	$e.l$	method invocation
	$\mid$	$e@\varphi$	object renaming
	$\mid$	$e \leftarrow\!\!+ l(s){=}e' : \tau$	object extension
	$\mid$	$e \leftarrow l(s){=}e'$	method override

calculus to a simple delegation-based system; objects, being extensible, also serve to encode classes. Variants of the calculus have been carefully studied before (e.g., [9, 10, 11, 12, 18]). To keep the setting simple, all objects are immutable and objects have no fields; fields can be encoded as methods which ignore their *self* argument.

2.1 Syntax

The language, whose syntax appears in Table 1, derives largely from the object calculi of Abadi and Cardelli [1], Fisher, Honsell, and Mitchell [10], and Liquori [18]. The types of the language include base types, function types, and object types. Object types only mention the visible names of methods and their return types. We identify object expressions or types differing only in the order of their components.

In most object calculi, objects draw their method names from a distinguished infinite set L of labels. Instead, our calculus distinguishes *internal* names from *external* names, and uses a dictionary to map between the two. For convenience, we choose the natural numbers $\mathbf{N}$ to serve as the internal names, L to serve as external names, and use l or m to denote elements of the set $L \cup \mathbf{N}$. The use of the natural numbers is meant to evoke "slots" in a method table or array, and makes the notation slightly simpler. There is, of course, nothing essential in using $\mathbf{N}$, or even in distinguishing the sets of external and internal labels.

A **dictionary** φ is a finite partial function from the extended set of labels $L \cup \mathbf{N}$ to $L \cup \mathbf{N}$. Dictionaries appear as subscripts on objects. For instance, the object

$$\mathsf{obj}\, s.\{\vert 1 \triangleright 3 : \mathsf{Int} \vert\}_{[x \mapsto 1]}$$

has the dictionary $[x \mapsto 1]$; when x is invoked, the actual code invoked is the method internally labeled by $[x{\mapsto}1](x) = 1$. We use $\varphi[l \mapsto n]$ to denote the partial function that behaves exactly as φ except for mapping l to n, and $id(S)$ to denote the identity function with finite domain $S \subseteq L \cup \mathbf{N}$.

There are three primitive operations on objects besides method invocation. The operation $e@\varphi'$, alters the existing dictionary on an object: it evaluates e to an object and composes φ' with its internal dictionary. In addition to renaming components, this operation can contract the number of methods visible in the object when the range of φ' is smaller than the domain of the dictionary on the object. For example,

$$\mathsf{obj}\, s.\{\vert 1 \triangleright 3 : \mathsf{Int}, 2 \triangleright (s@\varphi).x : \mathsf{Int} \vert\}_\varphi @[getx \mapsto getx]$$

where $\varphi = [x \mapsto 1, getx \mapsto 2]$, returns an object whose only visible method is *getx*, because the dictionary will be $[getx \mapsto 2]$. Similarly, one can increase the number of fields visible by mapping several external labels to the same internal label. (In this case, if one of these methods is overridden multiple methods may appear to change.) The other two operations add or change the methods of objects. The operation $e_0 \leftarrow\!\!+ l(s){=}e : \tau$ adds a new method l to the object denoted by e_0. The method expects a self parameter s, and when invoked evaluates the body e of type τ. An existing method can be replaced within an object by the operation $e_0 \leftarrow l(s){=}e$.

2.2 Static and dynamic semantics

To give dynamic semantics to the language, we use Felleisen's "evaluation context" formulation [8] of Plotkin's SOS [21]. The syntax of evaluation contexts (a subset of those expressions containing a single hole, denoted $[\cdot]$) is given by the grammar

$$\begin{array}{rl} E ::= & [\cdot] \mid (E\; e) \mid (v\; E) \\ \mid & E.l \\ \mid & E@\varphi \\ \mid & E \leftarrow\!\!+ l(s){=}e' : \tau' \\ \mid & E \leftarrow l(s){=}e' \end{array}$$

We write $E[e]$ to denote the evaluation context E with the hole replaced by e. The local reduction relation $\rightsquigarrow$ is shown in Table 2. These rules use a syntactic substitution operation, written $[s{\mapsto}e]\, e'$, which denotes the capture-free substitution of e for s in e'. The important point to note for this system is that the dictionary is stripped (and replaced by an identity dictionary) at method invocation. This allows method code to assume a default dictionary, however the object may be altered. Code for methods in method override and object extension is then coerced to expect such stripped objects. The relation in Table 2 is extended to a one-step evaluation relation on programs by

$$e \rightsquigarrow e' \quad \Longleftrightarrow \quad \exists e_1, e_2.\; e = E[e_1] \,\wedge\, e_1 \rightsquigarrow e_2 \,\wedge\, E[e_2] = e'.$$

We can prove

Proposition 1 (Determinacy)
The relation $\rightsquigarrow$ is a partial function.

We use $\rightsquigarrow^*$ to denote the reflexive, transitive closure of $\rightsquigarrow$.

The static semantics of the language is shown in Appendix A. The novel aspects are the subsumption rule for width subtyping—more precisely the fact that we allow naïve width subtyping on objects—and the treatment of dictionaries. The following theorem shows that the static semantics prevents run-time type errors:

Theorem 2 (Soundness)

1. *If $\vdash e : \tau$ and $e \rightsquigarrow e'$, then $\vdash e' : \tau$.*
2. *If $\vdash e : \tau$, then e is a value or $e \rightsquigarrow e'$ for some expression e'.*

2.3 Examples

The first example shows the behavior of private methods. Let $\varphi = [F{\mapsto}1, M{\mapsto}2]$ and define the explicit subtyping coercion $o{:}{>}\tau$ as

Table 2: Local Reduction Steps of First-Order System

$(\lambda x{:}\tau.e)\ v$	$\leadsto$	$[x{\mapsto}v]\,e$	
$(\mathsf{obj}\,s.\{\!|\,i{\triangleright}e_i:\tau_i\ ^{i\in 1..n}\,|\!\}_\varphi).l$	$\leadsto$	$[s{\mapsto}\mathsf{obj}\,s.\{\!|\,i{\triangleright}e_i:\tau_i\ ^{i\in 1..n}\,|\!\}_{id\{1,\dots,n\}}]\,e_{\varphi(l)}$	
$(\mathsf{obj}\,s.\{\!|\,i{\triangleright}e_i:\tau'_i\ ^{i\in 1..n}\,|\!\}_\varphi)@\varphi'$	$\leadsto$	$\mathsf{obj}\,s.\{\!|\,i{\triangleright}e_i:\tau'_i\ ^{i\in 1..n}\,|\!\}_{\varphi\circ\varphi'}$	
$(\mathsf{obj}\,s.\{\!|\,i{\triangleright}e_i:\tau_i\ ^{i\in 1..n}\,|\!\}_\varphi){\leftarrow\!\!+}l(s){=}e:\tau$	$\leadsto$	$\mathsf{obj}\,s.\{\!|\,i{\triangleright}e_i:\tau_i\ ^{i\in 1..n},(n+1){\triangleright}[s{\mapsto}s@\varphi']\,e:\tau\,|\!\}_{\varphi'}$	$\varphi'=\varphi[l{\mapsto}(n{+}1)]$
$(\mathsf{obj}\,s.\{\!|\,i{\triangleright}e_i:\tau_i\ ^{i\in 1..n}\,|\!\}_\varphi){\leftarrow}l(s){=}e$	$\leadsto$	$\mathsf{obj}\,s.\{\!|\,i{\triangleright}e_i:\tau_i\ ^{i\in(1..n)\setminus\varphi(l)},\varphi(l){\triangleright}[s{\mapsto}s@\varphi]\,e:\tau_{\varphi(l)}\,|\!\}_\varphi$	

shorthand for the term $((\lambda x{:}\tau.x)\ o)$. Consider the terms

$$o := ((\mathsf{obj}\,s.\{\!|\,|\!\}_{[]}){\leftarrow\!\!+}F(s){=}5:\mathsf{Int}){\leftarrow\!\!+}M(s){=}(s.F{+}1):\mathsf{Int}$$
$$o:\{\!|F:\mathsf{Int},M:\mathsf{Int}|\!\}$$
$$o\leadsto^*\mathsf{obj}\,s.\{\!|1{\triangleright}5:\mathsf{Int},2{\triangleright}(s@\varphi).F{+}1:\mathsf{Int}|\!\}_\varphi$$
$$o_1 := o{\leftarrow}F(s){=}7$$
$$o_1:\{\!|F:\mathsf{Int},M:\mathsf{Int}|\!\}$$
$$o_1\leadsto^*\mathsf{obj}\,s.\{\!|1{\triangleright}7:\mathsf{Int},2{\triangleright}(s@\varphi).F{+}1:\mathsf{Int}|\!\}_\varphi$$
$$o_2 := o{:>}\{\!|M:\mathsf{Int}|\!\}$$
$$o_2:\{\!|M:\mathsf{Int}|\!\}$$
$$o_2\leadsto^*\mathsf{obj}\,s.\{\!|1{\triangleright}7:\mathsf{Int},2{\triangleright}(s@\varphi).F{+}1:\mathsf{Int}|\!\}_\varphi$$
$$o_3 := o_2{\leftarrow\!\!+}F(s){=}\mathsf{True}:\mathsf{Bool}$$
$$o_3:\{\!|F:\mathsf{Bool},M:\mathsf{Int}|\!\}$$
$$o_3\leadsto^*\mathsf{obj}\,\mathsf{s}.\{\!|1{\triangleright}7:\mathsf{Int},2{\triangleright}(s@\varphi).F{+}1:\mathsf{Int},\ 3{\triangleright}\mathsf{True}:\mathsf{Bool}|\!\}_{[F\mapsto 3,M\mapsto 2]}$$

Here o has methods F and M. When a method is invoked, the self parameter s is replaced with an object with an identity dictionary. Thus, it is easy to see that $o.F$ evaluates to 5 and $o.M$ to 6. In o_1 we override F with a method that returns 7; $o_1.F$ evaluates to 7 and $o_1.M$ to 8. To obtain o_2, we use subsumption on o to make method F private, leaving only one visible method M. Then $o_2.M$ still evaluates to 6. The type system would reject any attempt to *override* F in o_2, since o_2 has no visible F method. It is legal, however, to *extend* o_2 to o_3 by adding a new method called F (which here happens to return a boolean value). The previous F method is still present in the underlying object; and evaluating $o_3.M$ still gives 6, while $o_3.F$ returns True.

As this example shows, under our semantics extending an object never changes the behavior of pre-existing methods. When a method is added to an object, we arrange for its body to invoke methods in *self* using internal labels. Its behavior does not change unless one of these is overridden (which cannot occur unless there is a corresponding external label).

This example also raises another point: object extension must be used carefully. One may always use extension in place of method override, but the consequences are different. For instance, consider the term

$$o_4 := o{\leftarrow\!\!+}F(s){=}7:\mathsf{Int}$$

which resembles o_1 except that we use extension rather than override. The term is typable because the object o is implicitly forced (via subsumption) to have an object type with only one method M so that o_4 evaluates to the same object as o_3. As such, $o_4.M$ returns 6 while $o_1.M$ returns 8. The programmer must be careful to determine which of these behaviors is correct and use the appropriate operation.

As a second example, consider the term

$$f := \lambda o{:}\{\!|x:\mathsf{Int}|\!\}.(o{\leftarrow\!\!+}getx(s){=}s.x:\mathsf{Int})$$

This function can be given the type

$$(\{\!|x:\mathsf{Int}|\!\}\to\{\!|x:\mathsf{Int},getx:\mathsf{Int}|\!\}).$$

In contrast to other formalisms, this function may be applied to *any* object with an x method of type Int, regardless of its other methods. On the other hand, there is some information loss: if we apply this function to an object with (public) methods x, y, and z, the result has just two public methods x and $getx$; y and z are hidden in the act of subsumption. An extension of the system with row variables [24] or bounded polymorphism [7] might allow preservation of methods in such cases.

3 Second-Order System

In a calculus of immutable objects, it is natural to consider objects that can return updated copies of themselves. For example, we might define a type of movable points, which could be defined (using a recursive type definition) as:

$$PT' := \{\!|getx:\mathsf{Int},move:\mathsf{Int}{\to}PT'|\!\}$$

where the *move* operation takes an amount to offset the position of the returned point. Now suppose we have such a point $pt':PT'$, and we extend it to a colored point by adding a *getc* method returning a color. The resulting object would have type

$$CPT' := \{\!|getx:\mathsf{Int},move:\mathsf{Int}{\to}PT',getc:\mathsf{color}|\!\}$$

Unfortunately, if $cpt':CPT'$ then $cpt'.move$ is a function which still returns a value of type PT; the color is lost.

This motivates a move to a "second-order calculus" in the parlance of [1]. Method types can now refer to "the type of self" which changes as the object is extended. Thus we define

$$PT := \mathsf{Obj}\,\alpha.\{\!|getx:\mathsf{Int},move:\mathsf{Int}{\to}\alpha|\!\}$$

where α represents the type of self, and is bound within the object type. Then the extension to add a color would have type

$$CPT := \mathsf{Obj}\,\alpha.\{\!|getx:\mathsf{Int},move:\mathsf{Int}{\to}\alpha,getc:\mathsf{color}|\!\}$$

Assuming $pt:PT$ and $cpt:CPT$, the method invocation $pt.move$ has type $[\alpha{\mapsto}PT]\,(\mathsf{Int}{\to}\alpha)=\mathsf{Int}{\to}PT$, and the method invocation $cpt.move$ has type $[\alpha{\mapsto}CPT]\,(\mathsf{Int}{\to}\alpha)=\mathsf{Int}{\to}CPT$ as desired.

Table 3: Syntax of the Second-Order System.

$\tau ::=$		b	base type
	$\mid$	α	type variable
	$\mid$	$\tau \to \tau'$	function type
	$\mid$	$\tau \Rightarrow \tau'$	dictionary type
	$\mid$	$\mathsf{Obj}\,\alpha.\{\!\mid l : \tau_l\ ^{l \in I} \mid\!\}$	object type
	$\mid$	$\mathsf{Obj}\,\alpha.\{\!\mid i : \tau_i\ ^{i \in 1..n} \mid\!\}_{\varphi}$	layout type
$\Gamma ::=$		$\bullet$	typing contexts
	$\mid$	$\Gamma, x{:}\tau$	
	$\mid$	$\Gamma, \alpha \preceq \tau$	
$v ::=$		c	constant
	$\mid$	$x, s, d, \ldots$	variable
	$\mid$	$\lambda x{:}\tau.e$	function
	$\mid$	φ	dictionary
	$\mid$	$\mathsf{obj}\,(s{:}\alpha).\{\!\mid i \triangleright e_i : \tau_i'\ ^{i \in 1..n} \mid\!\}_{\varphi}$	object
$e ::=$		v	value
	$\mid$	$e_1\ e_2$	application
	$\mid$	$e.l$	method invocation
	$\mid$	$e \,\&\, v$	dictionary replacement
	$\mid$	$e \leftarrow\!+ l(s{:}\alpha, d) = e' : \tau$	object extension
	$\mid$	$e \leftarrow l(s{:}\alpha, d) = e'$	method override
	$\mid$	$e ._v l$	invocation with dictionary
	$\mid$	$e \leftarrow_v l(s{:}\alpha, d) = e'$	override with dictionary

3.1 Syntax and static semantics

In parallel with the extension of syntax for object types, we also extend object values to bind a type variable α to represent the type of self within the object:

$$\mathsf{obj}\,(s{:}\alpha).\{\!\mid i \triangleright e_i : \tau_i\ ^{i \in 1..n} \mid\!\}_{\varphi}$$

(The entire syntax of the second-order language appears in Table 3.) Because of the possibility of object extension and width subtyping, there is no way to know when a method is typechecked exactly what the type of self will be when the method is invoked. Therefore, the standard technique for typechecking methods involving such self types is to require they be parametric in the type of *self*, so that they are guaranteed to work for *any* future version of the object; normally this is implemented by using bounded polymorphism.

Because we have dictionaries, we have a further complication: methods involving self types must work parametrically not only for any extension of the object, but for any change to the object's dictionary as well. For example, the code for *pt.move* and *cpt.move* above is the same by construction; however, *pt* and *cpt* will have different dictionaries, and the values returned by the *move* functions should have similarly different dictionaries. This causes difficulties for our first-order system, in which invoking a method discards the object's dictionary. (Another way of looking at the first-order semantics is that all methods added by extension or override start by replacing the dictionary on *self* with their own dictionary.)

The second-order calculus avoids such coercions by providing dictionaries as first-class values, and allowing operations to be *parameterized* by a dictionary. Both method lookup and method override can specify the particular dictionary to be used in finding the internal name for the method. Thus the code for the *move* method above will be able to access or override the *getx* method of *self* by using the dictionary appropriate for points *without* modifying the dictionary of *self*; *pt.move* will return a point and *cpt.move* will return a colored point.

We do not expect programmers to build dictionaries directly; the current dictionary of an object is supplied for use in bodies at method override or method extension time. As seen in Table 3, the syntax for method override and method extension binds not only a variable s representing *self* and a type variable α representing self type, but also a variable d which represents the dictionary of the object at the time the method is added. Method override and method invocation also may optionally specify a dictionary value. For example, the operation $e \leftarrow_v l(s{:}\alpha, d) = e'$ uses the dictionary v to find the method that needs to be replaced (namely, the one with internal label $v(l)$). In general, v is likely to be either a variable d or a constant dictionary.

The last operation we provide is dictionary replacement, written $e \,\&\, \varphi$, since there are occasions in which we still want to replace the dictionary of an object entirely. For simplicity, we do not provide the dictionary renaming operation of the first-order system, although it would not be difficult to add.

Not all dictionaries make sense in conjunction with all objects. If all one knows is the external names of an object, it is impossible to tell whether a particular dictionary is appropriate for that object. Therefore we extend the type system with **dictionary types** (written with $\Rightarrow$), and **layout types**. Layout types are object types exposing the inner layout of an object and information about its dictionary. The most precise type of an object is a layout type giving the types and internal names for all its components, and exposing the entries in its dictionary. Every layout type is a subtype of the corresponding object type derived from the component types and the dictionary specification. For instance, the layout type

$$\mathsf{Obj}\,\alpha.\{\!\mid 1 : \mathsf{Int}, 2 : \mathsf{Int} \to \alpha \mid\!\}_{[x \mapsto 1, move \mapsto 2]}$$

is a subtype of the object type

$$\mathsf{Obj}\,\alpha.\{\!\mid x : \mathsf{Int}, move : \mathsf{Int} \to \alpha \mid\!\}.$$

An object type is thus more abstract than any of its possible layout types.

Layout types are used in conjunction with dictionary types; a dictionary of type $\tau_1 \Rightarrow \tau_2$ can be used with objects of type τ_1 (usually a layout type or a type variable representing a layout type), and when used to replace that object's dictionary yields an object of type τ_2. In the second-order system, we can restrict dictionaries to finite, partial functions from L to $\mathbf{N}$.

The static semantics of the second-order calculus appears in Appendix B. The rules use the following abbreviations:

$$\top_{obj} := \mathsf{Obj}\,\alpha.\{\!\mid\,\mid\!\}$$
$$(\mathsf{let}\,x : \tau = e\,\mathsf{in}\,e') := ((\lambda x{:}\tau.e')\ e)$$

Here $\top_{obj}$ is the object type conveying the least information.

The typing rules resemble those of the first-order system, but with two significant additions. First, since the type of *self* is now a type variable, we need assumptions about the bounds of type variables. A new rule for this appears in the rules for well-formedness of contexts. Second, the system includes rules for the new operations and values. The rules for checking objects, method override, and method extension are the most complicated. These rules check the bodies of methods using assumptions about the type of *self* and (possibly) a dictionary d. In contrast to other systems in which the type α of *self* is a type variable bounded by some object type τ, we assume α is an abstract object type, and supply a dictionary d of type $\alpha \Rightarrow \tau$. This corresponds to the idea that in order to make real use of *self* as an object, we have to use a known dictionary rather than the one currently attached to *self*. For methods within object values, we have information about the underlying layout and hence can make self type a type variable bounded by a layout type.

A technical constraint is that the type α of *self* must appear *covariantly* inside layout and object types. We say that α appears **covariantly** in τ if any of the following is true:

- α is not free in τ;
- τ is α;
- τ is $\tau_1 \to \tau_2$ or $\tau_1 \Rightarrow \tau_2$, where α appears contravariantly in τ_1 and covariantly in τ_2;
- τ is $\mathsf{Obj}\,\alpha.\{\!|\, l : \tau_l\ ^{l \in I} |\!\}$ or $\mathsf{Obj}\,\alpha.\{\!|\, i : \tau_i\ ^{i \in 1..n} |\!\}_\varphi$, and α occurs covariantly in each τ_i.

Similarly, α appears **contravariantly** in τ if any of the following is true:

- α is not free in τ;
- τ is $\tau_1 \to \tau_2$ or $\tau_1 \Rightarrow \tau_2$, where α appears covariantly in τ_1 and contravariantly in τ_2;
- τ is $\mathsf{Obj}\,\alpha.\{\!|\, l : \tau_l\ ^{l \in I} |\!\}$ or $\mathsf{Obj}\,\alpha.\{\!|\, i : \tau_i\ ^{i \in 1..n} |\!\}_\varphi$, and α occurs contravariantly in each τ_i.

We would need more restrictive width subtyping to avoid unsoundness if the α were allowed to appear contravariantly (see [1] for examples). As such, this system does not handle binary methods (see [6] for a thorough discussion).

3.2 Dynamic semantics

The dynamic semantics for the second-order calculus uses evaluation contexts of the form

$$\begin{array}{rcl} E & ::= & [\cdot] \mid (E\ e) \mid (v\ E) \\ & \mid & E \,\&\, v \\ & \mid & E.l \\ & \mid & E._v l \\ & \mid & E \leftarrow\!\!+ l(s{:}\alpha, d) = e' : \tau' \\ & \mid & E \leftarrow l(s{:}\alpha, d) = e' \\ & \mid & E \leftarrow_v l(s{:}\alpha, d) = e' \end{array}$$

The rules for reducing redexes appear in Table 4; these rules define a relation $\leadsto$. As with the dynamic semantics of the first-order system, we write $e \leadsto e'$ when there is an evaluation context E such that $e = E[e_1]$, $e_1 \leadsto e_2$, and $e' = E[e_2]$. The dynamic semantics is deterministic:

Proposition 3 (Determinacy)
The relation $\leadsto$ is a partial function.

We again use the relation $\leadsto^*$ to denote the reflexive, transitive closure of $\leadsto$.

The static semantics is sound for the dynamic semantics:

Theorem 4 (Soundness)

1. *If $\vdash e : \tau$ and $e \leadsto e'$, then $\vdash e' : \tau$.*
2. *If $\vdash e : \tau$, then e is a value or $e \leadsto e'$ for some expression e'.*

3.3 Examples

Standard examples can now be written in the calculus. For instance, we can write code matching the above point and color point types:

$$\begin{array}{rcl} o & := & (\mathsf{obj}\,(s{:}\alpha).\{\!|\,|\!\}) \leftarrow\!\!+ getx(s{:}\alpha, d) = 0 : \mathsf{Int} \\ pt & ::= & o \leftarrow\!\!+ move(s : \alpha, d) = \\ & & \quad \lambda y{:}\mathsf{Int}.\mathsf{let}\ z = s._d getx \\ & & \qquad \mathsf{in}\ s \leftarrow_d getx(s'{:}\alpha', d') = z + y \\ cpt & ::= & pt \leftarrow\!\!+ getc(s{:}\alpha, d) = \mathsf{Red} : \mathsf{Color} \end{array}$$

There are many possible types for these objects besides PT and CPT. The most specific types are the layout types

$$\begin{array}{rl} o: & \mathsf{Obj}\,\alpha.\{\!|\, 1 : \mathsf{Int} \,|\!\}_{[getx \mapsto 1]} \\ pt: & \mathsf{Obj}\,\alpha.\{\!|\, 1 : \mathsf{Int}, 2 : \mathsf{Int} \to \alpha \,|\!\}_{[getx \mapsto 1, move \mapsto 2]} \\ cpt: & \mathsf{Obj}\,\alpha.\{\!|\, 1 : \mathsf{Int}, 2 : \mathsf{Int} \to \alpha, 3 : \mathsf{Color} \,|\!\}_{[getx \mapsto 1, move \mapsto 2, getc \mapsto 3]} \end{array}$$

These types are probably too detailed to report to the programmer, and might be suppressed by a real system. Instead, the following types are the most specific object types:

$$\begin{array}{rl} o: & \mathsf{Obj}\,\alpha.\{\!|\, getx : \mathsf{Int} \,|\!\} \\ pt: & \mathsf{Obj}\,\alpha.\{\!|\, getx : \mathsf{Int}, move : \mathsf{Int} \to \alpha \,|\!\} = PT \\ cpt: & \mathsf{Obj}\,\alpha.\{\!|\, getx : \mathsf{Int}, move : \mathsf{Int} \to \alpha, c : \mathsf{Color} \,|\!\} = CPT \end{array}$$

These types are supertypes of the above layout types.

The dynamic semantics of the language reduces these expressions to the following values:

$$\begin{array}{l} o \leadsto^* \mathsf{obj}\,(s{:}\alpha).\{\!|\, 1 \triangleright 0 : \mathsf{Int} \,|\!\}_{[getx \mapsto 1]} \\ pt \leadsto^* \mathsf{obj}\,(s{:}\alpha).\{\!|\, 1 \triangleright 0 : \mathsf{Int}, \\ \qquad 2 \triangleright (\lambda y{:}\mathsf{Int}.\mathsf{let}\ z = s._\varphi getx \\ \qquad\qquad \mathsf{in}\ s \leftarrow_\varphi getx(s'{:}\alpha', d') = z + y) : \\ \qquad\quad \mathsf{Int} \to \alpha \\ \qquad |\!\}_\varphi \\ cpt \leadsto^* \mathsf{obj}\,(s{:}\alpha).\{\!|\, 1 \triangleright 0 : \mathsf{Int}, \\ \qquad 2 \triangleright (\lambda y{:}\mathsf{Int}.\mathsf{let}\ z = s._\varphi getx \\ \qquad\qquad \mathsf{in}\ s \leftarrow_\varphi getx(s'{:}\alpha', d') = z + y) : \\ \qquad\quad \mathsf{Int} \to \alpha, \\ \qquad 3 \triangleright \mathsf{Red} : \mathsf{Color} \\ \qquad |\!\}_{[getx \mapsto 1, move \mapsto 2, getc \mapsto 3]} \end{array}$$

where $\varphi = [getx \mapsto 1, move \mapsto 2]$. Notice that the dictionary used in the body of the *move* method is φ, the one in place when the *move* method was added to the object. Thus, the *move* method looks in slot 1 for *getx*, and updates slot 1, even if a new method *getx* is later added to the object. The return type of *move* is still the type of *self*, so invoking *move* from both *pt* and *cpt* returns an object of the right type.

We can go a step further and define a form of classes for points and colored points, here instantiated as object-generating functions. For example,

$$\begin{array}{l} pt\_class := \lambda(x_0 : \mathsf{Int}). \\ \qquad (\mathsf{obj}\,(s{:}\alpha).\{\!|\,|\!\}_{[]} \\ \qquad \leftarrow\!\!+ x(s{:}\alpha, d) = x_0 : \mathsf{Int} \\ \qquad \leftarrow\!\!+ getx(s{:}\alpha, d) = s._d x : \mathsf{Int} \\ \qquad \leftarrow\!\!+ move(s{:}\alpha, d) = \\ \qquad\quad \lambda y{:}\mathsf{Int}.\mathsf{let}\ z = s._d getx \\ \qquad\qquad \mathsf{in}\ s \leftarrow_d x(s'{:}\alpha', d') = z + y \\ \qquad) :> PT \end{array}$$

$$\begin{array}{l} cpt\_class := \lambda(x_0{:}\mathsf{Int}).\,\lambda(c_0{:}\mathsf{Color}). \\ \qquad (pt\_class(x_0) \\ \qquad \leftarrow\!\!+ c(s : \alpha, d) = c_0 : \mathsf{Color} \\ \qquad \leftarrow\!\!+ getc(s{:}\alpha, d) = s._d c : \mathsf{Int}) :> CPT \end{array}$$

Here we have added the private field x, which is used by *getx* and *move*, but hidden from view by subsumption; *cpt_class* has type $\mathsf{Int} \to PT$. In particular, x is not in the scope of methods in the class *cpt_class* inheriting from *pt_class*. Similarly, we have added a private field c to the class of colored points, accessible only by the *getc* method, but through coercion the type of *cpt_class* is $\mathsf{Int} \to \mathsf{Color} \to CPT$. To typecheck and compile *cpt_class* we need only know the above type of *pt_class* which does not expose x.

Table 4: Local Reduction Steps for Second-Order System.

$(\lambda x{:}\tau.e)\ v$	$\rightsquigarrow$	$[x{\mapsto}v]\,e$	
$(\mathsf{obj}\,(s{:}\alpha).\{\mid i\triangleright e_i : \tau_i{}^{i\in 1..n}\mid\}_\varphi).l$	$\rightsquigarrow$	$[s{\mapsto}\mathsf{obj}\,(s{:}\alpha).\{\mid i\triangleright e_i : \tau_i{}^{i\in 1..n}\mid\}_\varphi]\,[\alpha{\mapsto}A]\,e_{\varphi(l)}$	$A = \mathsf{Obj}\,\alpha.\{\mid i : \tau_i{}^{i\in 1..n}\mid\}_\varphi$
$(\mathsf{obj}\,(s{:}\alpha).\{\mid i\triangleright e_i : \tau_i{}^{i\in 1..n}\mid\}_\varphi).{}_{\varphi'}l$	$\rightsquigarrow$	$[s{\mapsto}\mathsf{obj}\,(s{:}\alpha).\{\mid i\triangleright e_i : \tau_i{}^{i\in 1..n}\mid\}_\varphi]\,[\alpha{\mapsto}A]\,e_{\varphi'(l)}$	$A = \mathsf{Obj}\,\alpha.\{\mid i : \tau_i{}^{i\in 1..n}\mid\}_\varphi$
$(\mathsf{obj}\,(s{:}\alpha).\{\mid i\triangleright e_i : \tau_i{}^{i\in 1..n}\mid\}_\varphi)\,\&\,\varphi'$	$\rightsquigarrow$	$\mathsf{obj}\,(s{:}\alpha).\{\mid i\triangleright e_i : \tau_i{}^{i\in 1..n}\mid\}_{\varphi'}$	
$(\mathsf{obj}\,(s{:}\alpha).\{\mid i\triangleright e_i : \tau_i{}^{i\in 1..n}\mid\}_\varphi){\leftarrow}l(s{:}\alpha,d){=}e$	$\rightsquigarrow$	$\mathsf{obj}\,(s{:}\alpha).\{\mid i\triangleright e_i : \tau_i{}^{i\in(1..n)\setminus\varphi(l)}, \varphi(l)\triangleright[d{\mapsto}\varphi]\,e : \tau_{\varphi(l)}\mid\}_\varphi$	
$(\mathsf{obj}\,(s{:}\alpha).\{\mid i\triangleright e_i : \tau_i{}^{i\in 1..n}\mid\}_\varphi){\leftarrow_{\varphi'}}l(s{:}\alpha,d){=}e$	$\rightsquigarrow$	$\mathsf{obj}\,(s{:}\alpha).\{\mid i\triangleright e_i : \tau_i{}^{i\in(1..n)\setminus\varphi'(l)}, \varphi'(l)\triangleright[d{\mapsto}\varphi]\,e : \tau_{\varphi'(l)}\mid\}_\varphi$	
$(\mathsf{obj}\,(s{:}\alpha).\{\mid i\triangleright e_i : \tau_i{}^{i\in 1..n}\mid\}_\varphi){\leftarrow\!+}l(s{:}\alpha,d){=}e:\tau$	$\rightsquigarrow$	$\mathsf{obj}\,(s{:}\alpha).\{\mid i\triangleright e_i : \tau_i{}^{i\in 1..n}, (n{+}1)\triangleright[d{\mapsto}\varphi']\,e : \tau\mid\}_{\varphi'}$	$\varphi' = \varphi[l{\mapsto}(n{+}1)]$

This series of examples did not use dictionary replacement; this operation is useful when we want to delegate part of the action of a method to a predeclared function. For instance, define the function *getf* by:

$$getf := \lambda p{:}(\mathsf{Obj}\,\alpha.\{\mid F : \mathsf{Int}\mid\}).(p.f)$$

Then define the objects

$$\begin{aligned} o_1 := \quad & \mathsf{obj}\,(s{:}\alpha).\{\mid\mid\} \\ & {\leftarrow\!+}F(s{:}\alpha,d){=}4 : \mathsf{Int} \\ & {\leftarrow\!+}M(s{:}\alpha,d){=}getf(s\,\&\,d) : \mathsf{Int} \end{aligned}$$

$$o_1 : \mathsf{Obj}\,\alpha.\{\mid F : \mathsf{Int}, M : \mathsf{Int}\mid\}$$

$$\begin{aligned} o_2 := \quad & o_1 \\ & {\leftarrow\!+}F(s{:}\alpha,d){=}5 : \mathsf{Int} \\ & {\leftarrow\!+}N(s{:}\alpha,d){=}getf(s\,\&\,d) : \mathsf{Int} \end{aligned}$$

$$o_2 : \mathsf{Obj}\,\alpha.\{\mid F : \mathsf{Int}, M : \mathsf{Int}, N : \mathsf{Int}\mid\}$$

Then we have

$$\begin{array}{ll} o_1.F \rightsquigarrow^* 4 & o_2.F \rightsquigarrow^* 5 \\ o_1.M \rightsquigarrow^* 4 & o_2.M \rightsquigarrow^* 4 \\ & o_2.N \rightsquigarrow^* 5 \end{array}$$

Although $o_2.M$ and $o_2.N$ appear to have the same code, they evaluate to different values because the dictionaries they use have different views of which field corresponds to F. Recall also that although o_2 has an object type with three components, because we used extension rather than override for the second F field, the underlying representation has four methods.

4 Conclusions

4.1 Implementation issues

There is a tradeoff in using explicit dictionaries: dictionary manipulation may induce a run-time cost. In a setting where our object calculus is used directly, there are methods for modestly reducing the run-time costs of dictionaries. For example, in compiling the dictionary composition operation $e@\varphi'$, one can either choose to calculate the composition of e's dictionary φ with φ' directly, or calculate the composition *lazily* as the new object gets requests for methods. The former may be more efficient when there are frequent compositions and method invocations, the latter more efficient when there are fewer compositions.

Similarly, though it would be unsound to drop object components when they are hidden by subsumption, it would be possible to drop these components from the *dictionary*. By turning subsumption into a run-time coercion on dictionaries, an implementation can ensure that the order and position of entries in an object's dictionary always matches the static type; then dictionary lookups are guaranteed to take constant time. Whether this is a good idea depends on the frequency of subsumptions, and the cost of searching a dictionary of unknown size.

If one knows more about the style of programming in the calculus, more efficiencies can be gained. For instance, the calculus makes a good compilation target for single-inheritance class-based languages. In these languages, each class determines a "method table" that can be shared among all objects of the class (the fields of each object, of course, must be maintained separately). The mapping of method names to indices in the method table is the dictionary. Since the method table can be statically determined, method calls through *self* need not be matched to a slot in the method table: they can immediately jump to the method. That is, when φ is statically determinable, the compiler can do dictionary lookups at compile-time and not generate code involving this dictionary for $(e@\varphi).l$ in the first-order calculus or $e.{}_\varphi l$ in the second-order calculus.

Calls to methods from *outside* the method suite may still need to go through the dictionary, however. In Java, for instance, suppose we define two classes `A` and `B` and an interface `C` via the definitions

```
interface I {
  public int m (int x);
}
class A implements I {
  public int m (int x) { ... };
}
class B implements I {
  public int k (int x) { ... };
  public int m (int x) { ... };
}
```

In a context where a variable is known only to have type `I`, a method invocation of `m` must go through the dictionary: the variable could be an object from the class `A` (in which case `m` is the first method in the method table) or from the class `B` (in which case `m` is the second method in the method table).

In class-based languages, the only operations that create objects are constructor functions. Thus, when compiling such a language into our calculus, all of the object operations *except* method invocation can be confined to the constructor functions. Constructor functions first call their superclass constructor functions, which return a partially constructed object, and then add or override methods. If the superclass constructor is known—as it is in a language like Java—the dictionaries are known, and so substitutions and compositions of dictionaries can be done at compile time. Even in a language with parameterized classes, one can imagine doing much of the manipulation of dictionaries at *link time* when the base classes of parameterized classes become instantiated.

Any of the optimizations valid for untyped object-oriented languages should apply here as well.

4.2 Related work

Our calculus embodies solutions to two problems: providing a characterization of private methods, and the combination of subtyping and object extension. Previous work has attempted to address these problems, and it is worth comparing these solutions to ours.

In the context of modeling private components in objects, Fisher and Mitchell [13] give an account of private (as well as **protected**) methods and fields using abstract types. Abstract types can be used to hide the representations of objects from clients, even though the objects themselves have access to the internal representations. Information about the names of private fields and methods, however, is still exposed. Their account is in some sense more fundamental than ours: our calculus directly supports hiding, and does not attempt to describe it in more basic concepts. Rémy and Vouillon [23] consider a more direct account of private data in classes, but only as inlined constant values. In addition to not matching a standard implementation, their approach does not extend well to mutable fields in the presence of object cloning or functional update of objects. Eiffel [19] has operations for redefining and "undefining" the methods of a class, much like our single renaming operation does in the first-order calculus. We are not aware, however, of any formal accounts that establish the soundness of the Eiffel type system. Bracha and Lindstrom [5] define a coercive operation for hiding components of objects; this appears to behave similarly to our subsumption operation, at least for first-order objects. They formalize this operation within an untyped λ-calculus.

More work has addressed the problems with object extension and subtyping. Fisher and Mitchell [12], for instance, discuss the unsoundness of width subtyping in the presence of object extension. Their solution is to distinguish the types of objects which either method override and object extension (but no subtyping) from those which support width and depth subtyping but not method override or object extension. Later work has looked at other ways of combining width subtyping with object extension without losing soundness. Liquori [17, 18] gives first- and second-order systems in which the types of extensible objects list the names and types of (a superset of) methods hidden by subsumption; the types must match if the object is extended by a new method with the same name as a hidden method. The idea is related to an old idea: Jategaonkar and Mitchell [15] and Rémy [22] use types that keep track of which methods must be "absent" from an object. Bono, Bugliesi, Dezani, and Liquori [4, 3, 2] take a different approach: object types contain a conservative approximation of which methods each method invokes via *self*. A collection of methods can be forgotten via subsumption if no remaining methods might invoke a member of this collection. This is not useful, however, for the purposes of modeling private methods (which exist for the sole purpose of being used by public methods).

4.3 Future work

We have shown that there is a calculus with width subtyping and object extension, one that allows a general notion of strong privacy for fields and methods within classes. Whether we have chosen the best set of primitives to describe this behavior is, however, open to debate.

Given the choice of primitives, it appears that many extensions should be possible. For instance, it should be possible to add mutable fields and methods and allow imperative update rather than functional update. Variance annotations should also be simple to add to the calculus to support richer forms of subtyping. We are also interested in adapting this work to a setting in which classes are a primitive notion. Finally, it would be interesting to extend the language with bounded polymorphism, which would make the calculus more expressive. We do not anticipate any major difficulties in these directions. A much more difficult problem would be to extend our characterization of private data to systems in which self types are allowed to appear contravariantly.

Because of the relationship with implementation strategies for object-oriented languages, we expect the calculus (or some variant thereof) should be implementable in a fairly direct fashion. Nevertheless, as we discussed in Section 4.1, there are still a number of tradeoffs to be explored when implementing the full calculus.

Some other, more difficult problems arise, the most important of which is to find a better semantical framework for the calculus. Our proofs of type soundness were purely operational; what would be better is a deeper understanding of the calculus that would make the static semantic rules obvious. A translation of the calculus into a typed λ-calculus might shed some light, or a denotational framework might provide a better setting to evaluate different choices of static rules.

Acknowledgements: We thank Kathleen Fisher for several helpful conversations, Martin Odersky for pointing out the connections with Eiffel, and Viviana Bono, Perry Cheng, Gary Lindstrom, and the anonymous referees for comments on drafts.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[2] V. Bono, M. Bugliesi, M. Dezani, and L. Liquori. Subtyping constraints for incomplete objects. In *CAAP*, Lecture Notes in Computer Science. Springer-Verlag, 1997.

[3] V. Bono, M. Bugliesi, and L. Liquori. A lambda calculus of incomplete objects. In *Proceedings, Mathematical Foundations of Computer Science*, volume 1113 of *Lecture Notes in Computer Science*, pages 218–229. Springer-Verlag, 1996.

[4] V. Bono and L. Liquori. A subtyping for the Fisher-Honsell-Mitchell calculus of objects. In *Proceedings, Computer Science Logic 1994*, volume 933 of *Lecture Notes in Computer Science*, pages 16–30. Springer-Verlag, 1995.

[5] G. Bracha and G. Lindstrom. Modularity meets inheritance. In *Proceedings of the IEEE Computer Society International Conference on Computer Languages*, pages 282–290, Washington, DC, April 1992. IEEE Computer Society.

[6] K. B. Bruce, L. Cardelli, G. Castagna, The Hopkins Object Group, G. Leavens, and B. C. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):217–238, 1995.

[7] L. Cardelli and P. Wegner. On understanding types, data abstraction and parametric polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[8] M. Felleisen. The theory and practice of first-class prompts. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 180–190. ACM, 1988.

[9] K. Fisher. *Type Systems for Object-Oriented Programming Languages*. PhD thesis, Department of Computer Science, Stanford University, 1996.

[10] K. Fisher, F. Honsell, and J. C. Mitchell. A lambda calculus of objects and method specialization. *Nordic J. Computing (*formerly *BIT)*, 1:3–37, 1994. Preliminary version appeared in *Proc. IEEE Symp. on Logic in Computer Science,* 1993, 26–38.

[11] K. Fisher and J. C. Mitchell. A delegation-based object calculus with subtyping. In *Fundamentals of Computation Theory (FCT'95)*, number 965 in Lecture Notes in Computer Science, pages 42–61. Springer-Verlag, 1995.

[12] K. Fisher and J. C. Mitchell. The development of type systems for object-oriented languages. *Theory and Practice of Object Systems*, 1:189–220, 1996. Preliminary version appeared in *Proc. Theoretical Aspects of Computer Software,* Springer LNCS 789, 1994, 844–885.

[13] K. Fisher and J. C. Mitchell. On the relationship between classes, objects, and data abstraction. In *Proceedings of the International Summer School on Mathematics of Program Construction, Marktoberdorf, Germany*, Lecture Notes in Computer Science. Springer-Verlag, 1997. To appear. Revised version to appear in *Theory and Practice of Object Systems*.

[14] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[15] L. Jategaonkar and J. C. Mitchell. Type inference with extended pattern matching and subtypes. *Fundamenta Informaticae*, 19:127–166, 1993. Preliminary version appeared in the *Proceedings of the ACM Symposium on Lisp and Functional Programming*, 1988.

[16] J. Lakos. *Large-Scale C++ Software Design*. Addison-Wesley, 1996.

[17] L. Liquori. An extended theory of primitive objects: First and second order systems. Technical Report CS-23-96, Dipartimento di Informatica, Università di Torino, 1996.

[18] L. Liquori. An extended theory of primitive objects: First order system. In M. Aksit and S. Matsuoka, editors, *Proceedings of ECOOP-97, International European Conference on Object Oriented Programming*, number 1241 in Lecture Notes in Computer Science. Springer-Verlag, 1997.

[19] B. Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.

[20] J. C. Mitchell. Toward a typed foundation for method specialization and inheritance. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 109–124. ACM, 1990.

[21] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus Univ., Computer Science Dept., Denmark, 1981.

[22] D. Rémy. Type inference for records in a natural extension of ML. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming*, pages 67–95. MIT Press, 1994. An earlier version appeared in the *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, 1989.

[23] D. Rémy and J. Vouillon. Objective ML: A simple object-oriented extension of ML. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 40–53. ACM Press, 1997.

[24] M. Wand. Complete type inference for simple objects. In *Proceedings, Symposium on Logic in Computer Science*, pages 37–44. IEEE, 1987.

A Static Semantics of First-Order System

Well-formed Contexts $\boxed{\Gamma \vdash \diamond}$

$$\frac{}{\bullet \vdash \diamond} \quad (1)$$

$$\frac{\Gamma \vdash \tau}{\Gamma, x{:}\tau \vdash \diamond} \quad (2)$$

Well-formed Expressions $\boxed{\Gamma \vdash e : \tau}$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash c : \mathsf{typeof}(c)} \quad (3)$$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash x : \Gamma(x)} \quad (4)$$

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash (\lambda x{:}\tau.e) : (\tau \to \tau')} \quad (5)$$

$$\frac{\Gamma \vdash e : (\tau \to \tau') \qquad \Gamma \vdash e' : \tau}{\Gamma \vdash (e\,e') : \tau'} \quad (6)$$

$$\frac{\Gamma \vdash \tau \preceq \tau' \qquad \Gamma \vdash e : \tau}{\Gamma \vdash e : \tau'} \quad (7)$$

$$\frac{\Gamma \vdash e : \{\!| l : \tau |\!\}}{\Gamma \vdash e.l : \tau} \quad (8)$$

$$\frac{\Gamma \vdash e : \{\!| l : \tau_l{}^{\,l \in I} |\!\} \qquad \mathsf{Range}(\varphi) \subseteq \mathsf{I}}{\Gamma \vdash e@\varphi : \{\!| l : \tau_{\varphi(l)}{}^{\,l \in \mathsf{Dom}(\varphi)} |\!\}} \quad (9)$$

$$\frac{\begin{array}{c}\mathsf{Range}(\varphi) \subseteq \mathsf{1..n} \\ \forall i \in 1..n : \quad \Gamma, s{:}\{\!| j : \tau_j{}^{\,j \in 1..n} |\!\} \vdash e_i : \tau_i\end{array}}{\Gamma \vdash \mathsf{obj}\, s.\{\!| i \triangleright e_i : \tau_i{}^{\,i \in 1..n} |\!\}_\varphi : \{\!| l : \tau_{\varphi(l)}{}^{\,l \in \mathsf{Dom}(\varphi)} |\!\}} \quad (10)$$

$$\frac{\begin{array}{c} m \in I \\ \Gamma \vdash e : \{\!| l : \tau_l{}^{\,l \in I} |\!\} \qquad \Gamma, s{:}\{\!| l : \tau_l{}^{\,l \in I} |\!\} \vdash e'_m : \tau_m\end{array}}{\Gamma \vdash e \leftarrow m(s) = e'_m : \{\!| l : \tau_l{}^{\,l \in I} |\!\}} \quad (11)$$

$$\frac{\begin{array}{c} m \notin I \\ \Gamma \vdash e : \{\!| l : \tau_l{}^{\,l \in I} |\!\} \qquad \Gamma, s{:}\{\!| l : \tau_l{}^{\,l \in I}, m : \tau'_m |\!\} \vdash e'_m : \tau'_m\end{array}}{\Gamma \vdash e \leftarrow\!\!+ m(s) = e'_m : \tau'_m : \{\!| l : \tau_l{}^{\,l \in I}, m : \tau'_m |\!\}} \quad (12)$$

Well-formed Types $\boxed{\Gamma \vdash \tau}$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash b} \quad (13)$$

$$\frac{\Gamma \vdash \tau \qquad \Gamma \vdash \tau'}{\Gamma \vdash \tau \to \tau'} \quad (14)$$

$$\frac{\forall l \in I : \quad \Gamma \vdash \tau_l}{\Gamma \vdash \{\!| l : \tau_l{}^{\,l \in I} |\!\}} \quad (15)$$

Width Subtyping $\boxed{\Gamma \vdash \tau_1 \preceq \tau_2}$

$$\frac{\Gamma \vdash \tau}{\Gamma \vdash \tau \preceq \tau} \quad (16)$$

$$\frac{\Gamma \vdash \tau_1 \preceq \tau_2 \qquad \Gamma \vdash \tau_2 \preceq \tau_3}{\Gamma \vdash \tau_1 \preceq \tau_3} \quad (17)$$

$$\frac{\Gamma \vdash \tau'_1 \preceq \tau_1 \qquad \Gamma \vdash \tau_2 \preceq \tau'_2}{\Gamma \vdash \tau_1 \to \tau_2 \preceq \tau'_1 \to \tau'_2} \quad (18)$$

$$\frac{\Gamma \vdash \{\!| l : \tau_l{}^{\,l \in I \cup J} |\!\}}{\Gamma \vdash \{\!| l : \tau_l{}^{\,l \in I \cup J} |\!\} \preceq \{\!| l : \tau_l{}^{\,l \in I} |\!\}} \quad (19)$$

B Static Semantics of Second-Order System

Well-formed Contexts $\boxed{\Gamma \vdash \diamond}$

$$\frac{}{\bullet \vdash \diamond} \tag{20}$$

$$\frac{\Gamma \vdash \tau \qquad x \notin \mathsf{BV}(\Gamma)}{\Gamma, x{:}\tau \vdash \diamond} \tag{21}$$

$$\frac{\Gamma \vdash \tau \qquad \alpha \notin \mathsf{BV}(\Gamma)}{\Gamma, \alpha{\preceq}\tau \vdash \diamond} \tag{22}$$

Well-formed Expressions $\boxed{\Gamma \vdash e : \tau}$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash c : \mathsf{typeof}(c)} \tag{23}$$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash x : \Gamma(x)} \tag{24}$$

$$\frac{\Gamma, x{:}\tau \vdash e : \tau'}{\Gamma \vdash \lambda x{:}\tau.e : \tau \to \tau'} \tag{25}$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\, e_2 : \tau} \tag{26}$$

$$\frac{\Gamma \vdash e : \tau' \qquad \Gamma \vdash \tau' \preceq \mathsf{Obj}\,\alpha.\{\!| l : \tau |\!\}}{\Gamma \vdash e.l : [\alpha \mapsto \tau']\tau} \tag{27}$$

$$\frac{\Gamma \vdash e : \tau' \qquad \Gamma \vdash v : \tau' \Rightarrow \mathsf{Obj}\,\alpha.\{\!| l : \tau |\!\}}{\Gamma \vdash e._v l : [\alpha \mapsto \tau']\tau} \tag{28}$$

$$\frac{\Gamma \vdash e : \tau' \qquad \Gamma \vdash v : \tau' \Rightarrow \tau}{\Gamma \vdash e \& v : \tau} \tag{29}$$

$$\frac{\begin{array}{c}\Gamma \vdash \diamond \qquad \mathsf{Range}(\varphi) \subseteq 1..\mathsf{n} \\ \forall i \in 1..n : \quad \Gamma, \alpha{\preceq}\mathsf{Obj}\,\alpha.\{\!| j : \tau_j{}^{j \in 1..n} |\!\}_{[]}, s{:}\alpha \vdash e_i : \tau_i\end{array}}{\Gamma \vdash \mathsf{obj}\,(s{:}\alpha).\{\!| i \triangleright e_i : \tau_i{}^{i \in 1..n} |\!\}_\varphi : \mathsf{Obj}\,\alpha.\{\!| i : \tau_i{}^{i \in 1..n} |\!\}_\varphi} \tag{30}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash \tau \preceq \mathsf{Obj}\,\alpha.\{\!| m : \tau_m |\!\} \\ \Gamma, \alpha{\preceq}\top_{obj}, s{:}\alpha, d{:}\alpha \Rightarrow \tau \vdash e_2 : \tau_m\end{array}}{\Gamma \vdash e_1 {\leftarrow} m(s{:}\alpha, d){=}e_2 : \tau} \tag{31}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash v : \tau \Rightarrow \mathsf{Obj}\,\alpha.\{\!| m : \tau_m |\!\} \\ \Gamma, \alpha{\preceq}\top_{obj}, s{:}\alpha, d{:}\alpha \Rightarrow \tau \vdash e_2 : \tau_m\end{array}}{\Gamma \vdash e_1 {\leftarrow_v} m(s{:}\alpha, d){=}e_2 : \tau} \tag{32}$$

$$\frac{\begin{array}{c} m \notin I \\ \Gamma \vdash e_1 : \mathsf{Obj}\,\alpha.\{\!| l : \tau_l{}^{l \in I} |\!\} \\ \Gamma, \alpha{\preceq}\top_{obj}, s{:}\alpha, d{:}\alpha \Rightarrow \mathsf{Obj}\,\alpha.\{\!| l : \tau_l{}^{l \in I}, m : \tau |\!\} \vdash e_2 : \tau\end{array}}{\Gamma \vdash (e_1 {\leftarrow\!\!+} l(s{:}\alpha, d){=}e_2 : \tau) : \mathsf{Obj}\,\alpha.\{\!| l : \tau_l{}^{l \in I}, m : \tau |\!\}} \tag{33}$$

$$\frac{\Gamma \vdash \mathsf{Obj}\,\alpha.\{\!| i : \tau_i{}^{i \in 1..n} |\!\}_\varphi}{\Gamma \vdash \varphi : \mathsf{Obj}\,\alpha.\{\!| i : \tau_i{}^{i \in 1..n} |\!\}_{[]} \Rightarrow \mathsf{Obj}\,\alpha.\{\!| l : i : \tau_i{}^{i \in 1..n} |\!\}_\varphi} \tag{34}$$

$$\frac{\Gamma \vdash e : \tau' \qquad \Gamma \vdash \tau' \preceq \tau}{\Gamma \vdash e : \tau'} \tag{35}$$

Well-formed Types $\boxed{\Gamma \vdash \tau}$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash b} \tag{36}$$

$$\frac{\Gamma \vdash \diamond \qquad \Gamma = \Gamma', \alpha{\preceq}\tau, \Gamma''}{\Gamma \vdash \alpha} \tag{37}$$

$$\frac{\Gamma \vdash \tau_1 \qquad \Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 \to \tau_2} \tag{38}$$

$$\frac{\Gamma \vdash \tau_1 \qquad \Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 \Rightarrow \tau_2} \tag{39}$$

$$\frac{\begin{array}{c}\forall i \in 1..n : \quad \begin{cases} \Gamma, \alpha{\preceq}\top_{obj} \vdash \tau_i \\ \alpha \text{ occurs covariantly in } \tau_i \end{cases} \\ \Gamma \vdash \diamond \qquad l_1, \ldots, l_n \text{ distinct}\end{array}}{\Gamma \vdash \mathsf{Obj}\,\alpha.\{\!| l_i : \tau_i{}^{i \in 1..n} |\!\}} \tag{40}$$

$$\frac{\begin{array}{c}\forall i \in 1..n : \quad \begin{cases} \Gamma, \alpha{\preceq}\top_{obj} \vdash \tau_i \\ \alpha \text{ occurs covariantly in } \tau_i \end{cases} \\ \Gamma \vdash \diamond \qquad \mathsf{Range}(\varphi) \subseteq 1..\mathsf{n}\end{array}}{\Gamma \vdash \mathsf{Obj}\,\alpha.\{\!| i : \tau_i{}^{i \in 1..n} |\!\}_\varphi} \tag{41}$$

Width Subtyping $\boxed{\Gamma \vdash \tau_1 \preceq \tau_2}$

$$\frac{\Gamma \vdash \tau}{\Gamma \vdash \tau \preceq \tau} \tag{42}$$

$$\frac{\Gamma \vdash \diamond \qquad \Gamma = \Gamma', \alpha{\preceq}\tau', \Gamma'' \qquad \Gamma \vdash \tau' \preceq \tau}{\Gamma \vdash \alpha \preceq \tau} \tag{43}$$

$$\frac{\Gamma \vdash \tau_1' \preceq \tau_1 \qquad \Gamma \vdash \tau_2 \preceq \tau_2'}{\Gamma \vdash \tau_1 \to \tau_2 \preceq \tau_1' \to \tau_2'} \tag{44}$$

$$\frac{\Gamma \vdash \tau_1' \preceq \tau_1 \qquad \Gamma \vdash \tau_2 \preceq \tau_2'}{\Gamma \vdash \tau_1 \Rightarrow \tau_2 \preceq \tau_1' \Rightarrow \tau_2'} \tag{45}$$

$$\frac{\Gamma \vdash \mathsf{Obj}\,\alpha.\{\!| l : \tau_l{}^{l \in I \cup J} |\!\}}{\Gamma \vdash \mathsf{Obj}\,\alpha.\{\!| l : \tau_l{}^{l \in I \cup J} |\!\} \preceq \mathsf{Obj}\,\alpha.\{\!| l : \tau_l{}^{l \in I} |\!\}} \tag{46}$$

$$\frac{\begin{array}{c}\varphi' \subseteq \varphi \\ \Gamma \vdash \mathsf{Obj}\,\alpha.\{\!| i : \tau_i{}^{i \in 1..n+m} |\!\}_\varphi \qquad \mathsf{Range}(\varphi') \subseteq 1..\mathsf{n}\end{array}}{\Gamma \vdash \mathsf{Obj}\,\alpha.\{\!| i : \tau_i{}^{i \in 1..n+m} |\!\}_\varphi \preceq \mathsf{Obj}\,\alpha.\{\!| i : \tau_i{}^{i \in 1..n} |\!\}_{\varphi'}} \tag{47}$$

$$\frac{\Gamma \vdash \mathsf{Obj}\,\alpha.\{\!| i : \tau_i{}^{i \in 1..n} |\!\}_\varphi \qquad I \subseteq \mathsf{Dom}(\varphi)}{\Gamma \vdash \mathsf{Obj}\,\alpha.\{\!| i : \tau_i{}^{i \in 1..n} |\!\}_\varphi \preceq \mathsf{Obj}\,\alpha.\{\!| l : \tau_{\varphi(l)}{}^{l \in I} |\!\}} \tag{48}$$