Imperative Objects with Dependent Types

Joana Campos
Lasige and Department of Informatics
University of Lisbon, Portugal
jcampos@lasige.di.fc.ul.pt

Vasco T. Vasconcelos
Lasige and Department of Informatics
University of Lisbon, Portugal
vv@di.fc.ul.pt

ABSTRACT

Index refinements (or dependent types over a restricted domain) enable the expression of many desirable invariants that can be verified at compile time. We propose to incorporate a system of index refinements in a small, class-based, imperative, object-oriented language. While rooted in techniques formulated for dependently-typed functional languages, our type system is able to capture more than just value properties and pure computations. Index refinements, combined with a notion of pre- and post-type (i.e. a type that describes the initial and final state of an object), give programmers the ability to reason about effectful computations. Our type system distinguishes between two classes of objects, imposing an affine discipline to objects whose types are governed by indices, as opposed to conventional objects which can be freely shared. We have designed and implemented an expressive and decidable type system, which we illustrate through a number of examples.

Keywords

Dependent types, index refinements, classes, mutable objects

1. INTRODUCTION

Extending the concepts of dependent types to express and statically verify invariants in imperative, object-oriented programming is appealing. Since Xi and Pfenning [16], there has been a considerable interest in the formulation of index refinement systems that allow programmers to specify and statically check many desirable properties. However, with few exceptions, most systems provide frameworks that capture properties in pure computations, rather than incorporating effects (mutable state).

This paper presents Dependent Object-oriented Language (DOL), a new programming language that allows programmers to refine types in order to reason about properties of mutable objects. The language provides a type system equipped with (1) index refinements using a separate (pure) language of indices to describe a variety of properties that cannot be captured by conventional types, and (2) a notion of pre-[1] and post-type that allows programmers to describe the state of parameters at method entry and exit. The two features combined enable static checking of pre- and post-states of objects without the need of introducing separate Hoare-style specifications. For programmers, this type-based approach has both the advantage of reducing the annotation burden and relieving them from the task of guiding the verification process.

We follow the approach found in Dependent ML (DML) [16], whereby types are parametric on indices drawn from a separate language of constraints, rather than providing full dependency on terms of arbitrary types. The result is still an expressive type system in which typechecking is decidable as long as indices fall within a decidable theory. In this paper, we give examples using the integer domain, which is by far the most explored (decidable) constraint theory. Exploring the integration of other more expressive domains, possibly at the cost of typechecking decidability, is one of the directions we point to in future work.

In DOL, we adopt the "pure" object-oriented programming model, similar to the Smalltalk or Ruby object models, in order to obtain a simple type system; hence, (1) object references are the only values in our language (as opposed to, say, Java that includes primitive values), (2) methods use a call-by-value strategy only (where values are object references), and (3) state-modifying methods which do not return values (similar to void methods) are the only kind of methods that can be defined in DOL.

Our language allows mutable, unique objects, as well as shared objects, which can be used in an unrestricted fashion, since some aliasing is required in practical object-oriented programming. For this paper, DOL takes the simplest approach to ownership control by imposing safe destructive updates on objects whose types are governed by indices. Shared objects have concrete, conventional types. For these, the type system does not provide any guarantee. More advanced approaches that handle these concerns exist in the literature, and, if desired, could be added to DOL. We identify one less restrictive approach in future work.

Contributions. The main contributions of this work include: a formulation of universal and existential dependent types in a class-based, imperative, object-oriented language; the introduction of a notion of pre- and post-type, allowing the type system to track mutable state in method parame-

[1] Other uses of the term "pre-type" exist in the literature, namely by Walker [13], which are unrelated to ours.

ters; support for single inheritance whereby a class may extend another class as long as it satisfies the index constraints defined by the superclass (it also means that programmers can define inductive structures in DOL); the definition of a type system for objects that relies on a decidable domain of indices and on a simple notion of unique ownership.

Outline. In Section 2, we discuss related work. Section 3 presents some motivating examples. We introduce the formal language in Section 4, and provide some of the rules in our type system in Section 5. Finally, we conclude in Section 6, outlining future work.

2. RELATED WORK

On the basis of index refinements is the notion of dependent type developed by Martin-Löf [8], used in proof assistants, like Coq [4], and by a number of languages, such as Cayenne [1], Epigram [9], and Idris [3]. With full dependent types, determining whether two types are equal comes down to determining term equivalence, which is undecidable in general. These languages provide different ways to handle non-terminating programs. The Cayenne system, for example, becomes semi-decidable by forcing the type-checker to terminate within a number of prescribed steps. The Hoare Type Theory (HTT) [10] proposes to extend a full dependently-typed functional language with an indexed monadic type in the style of a Hoare triple. While our pre- and post-types have similarities with the Hoare type, our work targets the Java-like paradigm, and does not involve the complexity of higher-order abstraction.

Index refinements as formulated by Xi and Pfenning in DML [16] reduce typechecking to a constraint satisfaction problem on terms belonging to index sorts. Their approach, which is also ours, offers the additional advantage of relative simplicity of the type system, as well as requiring fewer annotations, when compared to full dependent type systems. Xi later formulated Xanadu [15], a language with a C-like syntax combining imperative programming with index refinements. While closely related, DOL extends the ideas of Xanadu to a class-based language, which exhibits a rigid class structure and membership, provides inheritance and subtyping, and allows reasoning about mutable objects.

X10's constrained types [11] can also be considered a form of refinement types. These types are designed around the notion of constraints on the *immutable* state of objects. The core language proposed (CFJ) extends the purely functional FJ [6]. To our knowledge, support for reasoning about mutable objects and effects has not yet been provided.

Other approaches include advanced techniques for verifying software properties. Assertions are one of the most useful mechanisms, also serving for documenting code. Any boolean predicate can be used to check software properties, which makes statically checking assertions undecidable in general. The pre- and post-types in DOL can be seen as a (decidable) alternative to a pair of assertions.

ESC [7] and related systems are effective in finding bugs in Java programs. ESC provides a simple language of annotations to help detect errors at compile time, but the tool is based on an approach which is not sound nor complete. However, by combining a range of techniques, it is more expressive than index refinements. Spec♯ [2] uses a sound programming methodology to check C♯ programs, which allows reasoning about object invariants even in a multi-threaded setting. In DOL, we explore index refinements in an imperative, object-oriented language, requiring simple type annotations. These advanced techniques could in principle be accomodated in our language.

```
1  class Account(b:natural) ⇒
2    init          :: Empty..Account 0
3    deposit       :: m:natural ⇒
4                     Account b..Account (b+m), Integer m
5    withdraw      :: m:natural ⇒
6                     Account (b+m)..Account b, Integer m
7    getBalance :: x:integer ⇒ Account b, Integer x..Integer b
8    balance       :: Integer b // the only field
9    init()    = balance := 0
10   deposit(amount) = balance.add(amount)
11   withdraw(amount) = balance.sub(amount)
12   getBalance(bal) = bal.copy(balance)
13
14 class SavingsAccount(n:natural, s:{x:natural | x ≥ n})
15                                        extends Account s ⇒
16   init          :: Empty..SavingsAccount (n,s),
17                    Integer n, Integer s
18   withdraw      :: p:natural ⇒
19                    Account (s+p)..SavingsAccount (n,s), Integer p
20   minBalance :: Integer n // another field
21   // implementation omitted
```

Figure 1: A bank account and a subclass

3. MOTIVATING EXAMPLES

Classes and methods in DOL are parametric on the underlying index language, which refines object types. Dependencies are restricted to terms of index sorts. For example, in the domain of integers, index sorts comprise the integers and subsets of index sorts. In the examples, we often use the sort **natural**, which abbreviates {x:**integer** | x ≥ 0}. We write (⇒) to denote the mapping of indices to types. Note that classes and methods are parametric on universally quantified variables, although we omit the **all** quantifier in the examples; however, we signal the introduction of existentially quantified variables with **exists**.

Like Java, DOL supports single class inheritance. The default superclass Empty, provided by the language, is a concrete class which has no fields or methods, except for init. Instances of this class can be freely shared. In the examples, we often omit the **extends** declaration. By convention, init is the name of the method immediately called after object creation to initialise all fields of a class – including the inherited ones; it is the closest we have to a constructor.

3.1 Example 1: A Bank Account

Figure 1 shows the Account class. It is parametric on index variable b of sort **natural** denoting its current balance. The class exposes its interface in lines 2–8, defining the types of each class member. In the formal language, the structural and nominal types are used interchangeably (Account being a type family).

A method type declares a list of typed indices, a comma-separated list of parameter types, and no return type (DOL has only "void", state-modifying methods). The first parameter type by default describes the method's receiver, which we often call **this**. Each parameter type has two components, separated by dots (..). The first component is the pre-type that specifies the exact value of the argument at method entry. The second component is the post-type that specifies the argument's result value. Often, pre- and post-types are the same, either because the argument state does not change, or because the argument is a conventional object, which the type system does not track (as opposed to instances of type families). In the examples, we adopt the convention of omitting the post-type rather than duplicating it. For instance, the deposit type defined in lines 3–4 is equivalent to the full type:

```
deposit :: m:natural ⇒
           Account b..Account (b+m), Integer m..Integer m
```

where Account b..Account (b+m) qualifies **this** and Integer m.. Integer m qualifies the method's only argument. As a precondition, the pre-type states what the method requires. As a postcondition, the post-type states the produced state.

State Modifying Methods. Pre- and post-types impose a sort of usage behaviour. Informally, the Account interface allows an account to be created with no balance and requires that some funds are deposited before an amount, less or equal to the current balance, can be withdrawn. The account's current balance can be obtained at all times.

We now explain in detail the class interface (lines 2–8), sometimes referring to method implementations (lines 9–12). The init constructor returns Account 0 (a type family application) to describe the newly created object, and requires no existing state (we use Empty at the pre-type placeholder). deposit can be called on an object of type Account b and changes its state to Account (b+m); it takes some object amount and does not change its state. The withdraw method does, in some sense, the reverse operation: it changes the receiver's initial state from Account (b+m) to Account b; it takes some amount object and does not change it. Finally, getBalance does not change the receiver's state, but it takes some bal referencing an arbitrary integer object (note that the type family Integer is different from the sort **integer**), and changes its state to Integer b. Even though getBalance does not change the state of the receiver object, the method is not a "getter" in the sense of Java, since it produces a side effect on its argument.

We can now see how the typechecker catches a usage violation on client code that creates an account and calls some methods on it. We note, on each line, the type environment configuration:

```
var a := new Account(); // a: Account 0
a.deposit(100);         // a: Account 100
a.withdraw(70);         // a: Account 30
var v := 0;             // a: Account 30, v: Integer 0
a.getBalance(v);        // a: Account 30, v: Integer 30
a.withdraw(50)          // Error: 50 > 30
```

Any method implementation (not just client code) must be correct up to the detail included in the method signature. For example, the getBalance body (line 12) is typechecked starting from an initial type environment, where **this** and bal are given the defined pre-types (line 7), and ending with a (different) final environment where the types of **this** and bal are required to conform to the defined post-types.

Base Types. DOL does not distinguish between values which are objects and values of primitive types, typically found in Java-like languages. However, it provides native classes, such as Integer that defines a set of useful methods, including add, sub, or copy that makes a copy of the internal state of the given object to the receiver. Technically, the program constant 0 (line 9), distinct from the index value 0 (line 2), is syntactic sugar for the object reference returned by **new** Integer(), whereas the number 3 is short for a reference t such that t := **new** Integer(); t.inc(); t.inc(); t.inc().

```
1  class Node ⇒ {}
2  class Nil extends Node ⇒
3   init      :: Empty..Nil
4  class Cons extends Node ⇒
5   init      :: Empty..Cons, Node, Transaction
6   next      :: Node          // fields
7   value     :: Transaction
8   // implementation omitted
9  class SList(n:natural) ⇒
10  init      :: Empty..SList 0
11  addFirst  :: SList n..SList (n+1), Transaction
12  remFirst  :: SList (n+1)..SList n
13  get       :: i:{x:natural | x<n} ⇒
14               SList n, Integer i, Transaction
15  filter    :: SList n, Predicate,
16        SList 0..(exists m:{x:natural | x≤n} ⇒ SList m)
17  head      :: Node          // fields
18  count     :: Integer n
19  // implementation omitted
```

Figure 2: A dependently-typed singly linked list

Indexed Classes and Subtyping. Class SavingsAccount (Figure 1) extends Account. It is parametric on two index variables (line 14): the additional index n of type **natural** denotes a minimum opening deposit and balance for the account, constraining the type of index s, which keeps track of the account's current balance. Declaring that SavingsAccount **extends** Account s allows the typechecker to determine the relationship between the index variables of the subclass and the index variables of its direct superclass.

A class inherits from its superclass all the fields and methods that are not overridden. A method in a subclass is allowed to override a method with the same name that is present in the superclass. In the example, the typechecker verifies that the withdraw type in the subclass (lines 18–19) is a valid subtype of the corresponding method type in the superclass (lines 5–6). The notion of subtyping of methods relies on the usual contravariance of the argument's initial type and covariance of the resulting one.

Controlled Aliasing. The potential sources of aliasing problems in DOL are assignment and parameter passing. Given a: Account 30 and another reference a2, it is easy to show that the code a2 := a; a2.withdraw(30); a.withdraw(5) would break the invariant of object a, since the type system is no longer able to use the object's type to keep track of its state. To deal with assignment, we disallow aliasing of objects whose type is governed by indices, making object a acquire type Empty after its value is "written" to a2. However, since state changes are made explicit in method signatures, we do not need to empty an object in parameter passing (when it is just "read").

3.2 Example 2: A Singly Linked List

We introduce in Figure 2 a singly linked list of transactions. The SList class is parametric on index n of type **natural** used to track the list size and provide safe methods. Informally, the class interface (lines 10–18) specifies that a list is created empty, and that methods remFirst and get may only be called if an item was first added to the list.

Shared Types. Inheritance and shared types allow us to

$$
\begin{array}{rcll}
P & ::= & L_1 \ldots L_n & \text{(programs)}\\
L & ::= & \mathsf{class}\ C : \Pi\Delta \lhd C\bar{x}.T\ \mathsf{is}\ M & \text{(classes)}\\
M & ::= & \{l_k(y_k) = t_k{}^{k\in 1\ldots n}\} & \text{(methods)}\\
T & ::= & C \mid \{l_k\colon T_k{}^{k\in 1\ldots n}\} & \\
 & & \mid \Pi x : I.T \mid \Sigma x : I.T \mid Ti & \\
 & & \mid T \times T \mid T..T & \text{(types)}\\
t & ::= & s \mid y.l \coloneqq s \mid y.l \coloneqq (\mathsf{new}\ C)\bar{s} & \\
 & & \mid (s.l)s \mid \mathsf{if}\ s = s\ \mathsf{then}\ t\ \mathsf{else}\ t & \\
 & & \mid \mathsf{while}\ s = s\ \mathsf{do}\ t \mid t;t & \text{(terms)}\\
s & ::= & y \mid y.l & \text{(references)}\\
I & ::= & \mathsf{integer} \mid \{x\colon I \mid p\} & \text{(index types)}\\
i & ::= & x \mid n \mid i+i \mid \mathsf{max}(i,i) & \text{(index terms)}\\
p & ::= & i \le i \mid p \wedge p & \text{(propositions)}\\
\Delta & ::= & \epsilon \mid \Delta, x : I & \text{(index contexts)}
\end{array}
$$

Figure 3: User's syntax

$$
\frac{}{\mathsf{ctype}(\mathsf{Empty}) = \{\mathsf{init} : \mathsf{Empty}\}}
$$

$$
\frac{\mathsf{class}\ C : \Pi\Delta \lhd D\bar{x}'.T\ \mathsf{is}\ M \qquad \Delta = \bar{x} : \bar{I}, \bar{x}' : \bar{J} \qquad \mathsf{ctype}(D) = \Pi\Delta'.U \qquad \bar{x}' : \bar{J} \vdash \theta : \Delta'}{\mathsf{ctype}(C) = \Pi\Delta.(T + (U\theta)[{}^{C\bar{x}}/{}_{D}])}
$$

$$
\frac{C.l = U \text{ implies } \epsilon \vdash T <: U}{\mathsf{override}(C, l, T)}
$$

Figure 4: Auxiliary definitions

write singly linked lists inductively in terms of the Nil and Cons classes, derived from the Node abstract class. In DOL, a class without an init method is like an abstract class in Java, since it cannot be instantiated. Rather than use **null** which is not a value in DOL, we adopt a functional style approach. The list is defined using field head of the shared type Node, denoting all possible forms of nodes. An empty list is one in which head references an instance of Nil, while in a non-empty list the field references an instance of Cons, containing two fields: one that refers to a transaction and another one that refers to the next node.

Existential Types. All the types that we have seen so far are universally quantified, since these are by far the most common. The filter signature (lines 15–16) provides an example of an existential type. The method must be called on an object of type SList n, which does not change. It takes a predicate of type Predicate, and an empty list that changes to a state described by type (**exists** m:{x:**natural** | x ≤ n} ⇒ SList m). The existential quantifier can only constrain the resulting list size to be smaller or equal to the current list size, since its exact size is unknown.

4. FORMAL LANGUAGE

We present the user's syntax in Figure 3. To represent universally quantified types of classes and methods in the internal syntax, we use Π as a dependent product restricted to indices. The existential type takes the form of dependent sums, with Σ quantifying existentially over index variables. A class has a structural representation as a dependent record

$\boxed{\Delta \vdash T}$

$$
\frac{\mathsf{class}\ C : \Pi\Delta' \lhd D\bar{x}.T\ \mathsf{is}\ M}{\Delta \vdash C}\ (\textsc{Wf-Class})
$$

$$
\frac{\forall\ 1 \le k \le n \qquad \Delta \vdash T_k}{\Delta \vdash \{l_k\colon T_k{}^{k\in 1\ldots n}\}}\ (\textsc{Wf-Record})
$$

$$
\frac{\Delta \vdash I \qquad \Delta, x : I \vdash T}{\Delta \vdash \Pi x : I.T}\ (\textsc{Wf-}\Pi)
$$

$$
\frac{\Delta \vdash I \qquad \Delta, x : I \vdash T}{\Delta \vdash \Sigma x : I.T}\ (\textsc{Wf-}\Sigma)
$$

$$
\frac{\Delta \vdash T <: \Pi x : I.T' \qquad \Delta \vdash i : I}{\Delta \vdash Ti}\ (\textsc{Wf-App})
$$

$$
\frac{\Delta \vdash T_1 \qquad \Delta \vdash T_2}{\Delta \vdash T_1 \times T_2}\ (\textsc{Wf-}\times)
$$

$$
\frac{\Delta \vdash T_1 \qquad \Delta \vdash T_2}{\Delta \vdash T_1..T_2}\ (\textsc{Wf-..})
$$

Figure 5: Rules for type formation

type of the form $\Pi\Delta.T$, where Δ is a context of typed index variables and T represents a record of member types. A class declaration class $C : \Pi\Delta \lhd D\bar{x}.T$ is M introduces a class named C that extends ($\lhd$) a superclass $D\bar{x}$, where the variables in $\bar{x}$ are declared in Δ. Methods are defined in M. Terms are fairly standard, including references, assignment to fields, method calls, the conditional, the loop, and the sequential term composition. We omit local variables found in the examples. We always use the letter y to distinguish object identifiers from index identifiers x, used only in types. To simplify the typing rules, every method declares exactly one parameter, except for the constructor that, in the spirit of FJ [6], always takes one parameter for each field, including all the inherited ones.

Types. A type classifies a class, a method or an object. It can be of the following seven forms:

- A class name C is a nominal object type induced by classes, commonly found in most mainstream object-oriented languages. We permit its use as a *type alias* [12] for the structural representation of a parametrised record type.
- A record type $\{l_k\colon T_k{}^{k\in 1\ldots n}\}$ exposes the class members. Nominal types are inherently recursive, therefore C may occur in any T_j for $1 \le j \le n$. The class name provides for the recursion fixed point.
- A type family $\Pi x : I.T$ is a type that maps elements of index type I to elements of the main type T, where x may occur free in T. It can be used to build up class and method types.
- An existential type $\Sigma x : I.T$ classifies objects of type T where x of index type I represents some unknown value in T.
- A type application Ti instantiates a type family.
- A pair of types of the form $T \times T$ is used for parameters, where the first one classifies this, implicitly passed to the method.
- A parameter type of the form $T..T$ defines a relationship between two components, the pre-type and a pos-

$\boxed{\Delta \vdash T <: U}$

$$\frac{}{\Delta \vdash C <: \mathsf{ctype}(C)}\ (\text{S-ClassL}) \qquad \frac{}{\Delta \vdash \mathsf{ctype}(C) <: C}\ (\text{S-ClassR}) \qquad \frac{}{\Delta \vdash T <: T}\ (\text{S-Refl})$$

$$\frac{\mathsf{class}\ C : \Pi(\bar{x} : \bar{I}, \bar{x}' : \bar{J}) \lhd D\bar{x}'.T\ \mathsf{is}\ M \qquad \Delta \vdash \bar{i} : \bar{I}}{\Delta \vdash C\bar{i} <: D}\ (\text{S-Sub}) \qquad \frac{\forall 1 \le k \le n \qquad \Delta \vdash T_k <: U_k}{\Delta \vdash \{l_k : T_k{}^{k \in 1\ldots n+m}\} <: \{l_k : U_k{}^{k \in 1\ldots n}\}}\ (\text{S-Record})$$

$$\frac{\Delta \vdash T[{}^{i}/_{x}] <: U \qquad \Delta \vdash i : I}{\Delta \vdash \Pi x : I.T <: U}\ (\text{S-}\Pi\text{L}) \qquad \frac{\Delta, x : I \vdash T <: U}{\Delta \vdash T <: \Pi x : I.U}\ (\text{S-}\Pi\text{R}) \qquad \frac{\Delta \vdash i : I}{\Delta \vdash (\Pi x : I.T)i <: T[{}^{i}/_{x}]}\ (\text{S-}\beta)$$

$$\frac{\Delta, x : I \vdash T <: U}{\Delta \vdash \Sigma x : I.T <: U}\ (\text{S-}\Sigma\text{L}) \qquad \frac{\Delta \vdash T <: U[{}^{i}/_{x}] \qquad \Delta \vdash i : I}{\Delta \vdash T <: \Sigma x : I.U}\ (\text{S-}\Sigma\text{R}) \qquad \frac{\Delta \vdash T <: U \qquad \Delta \models i \doteq j}{\Delta \vdash Ti <: Uj}\ (\text{S-App})$$

$$\frac{\Delta \vdash T_1 <: U_1 \qquad \Delta \vdash T_2 <: U_2}{\Delta \vdash (T_1 \times T_2) <: (U_1 \times U_2)}\ (\text{S-}\times) \qquad \frac{\Delta \vdash U_1 <: T_1 \qquad \Delta \vdash T_2 <: U_2}{\Delta \vdash (T_1..T_2) <: (U_1..U_2)}\ (\text{S-..})$$

$$\frac{\Delta \vdash T_1 <: T_2 \qquad \Delta \vdash T_2 <: T_3}{\Delta \vdash T_1 <: T_3}\ (\text{S-Trans})$$

Figure 6: Subtyping rules

$\boxed{\vdash L}$

$$\frac{\epsilon \vdash \Delta \qquad \Delta \vdash D\bar{x}, T \qquad \mathsf{this} : C \vdash_D M}{\vdash \mathsf{class}\ C : \Pi\Delta \lhd D\bar{x}.T\ \mathsf{is}\ M}\ (\text{T-Class})$$

$\boxed{\Gamma \vdash_D l(y) = t}$

$$\frac{\begin{array}{c} C.l = \Pi\Delta.(T_1..T_2) \times (U_1..U_2) \qquad \mathsf{override}(D, l, C.l) \\ \Delta; \mathsf{this} : T_1, y : U_1 \vdash t : \mathsf{Empty} \dashv \mathsf{this} : T_3, y : U_3 \\ \Delta \vdash T_1, T_3, U_3 <: \mathsf{ctype}(C), T_2, U_2 \end{array}}{\mathsf{this} : C \vdash_D l(y) = t}\ (\text{T-Meth})$$

Figure 7: Rules for program formation

sibly different post-type, representing the state of an object at method entry and exit.

Index Constructs. We include only a subset of the possible index constructs: variables, integer literals, addition of index terms, and a function that returns the greater of two index terms. Index types I comprise the integer type and the refinement type of the form $\{x : I \mid p\}$, constrained by a predicate p. Context Δ maps index variables to index types, and is used as a type environment of index identifiers.

5. TYPE SYSTEM

For space reasons, we have omitted from this paper a number of details related to index binding and substitution in types (extended pointwise to multiple substitution θ), as well as proofs of soundness via subject reduction and progress properties.

Static Semantics. We typecheck our language with respect to two type environments, Δ (already mentioned), and Γ mapping object identifiers to ordinary types. The main judgements consist of $\Delta \vdash I$ and $\Delta \vdash T$ for checking type formation, $\Delta \vdash I <: J$ and $\Delta \vdash T <: U$ for checking the subtype relation, and $\Delta \vdash i : I$ and $\Delta; \Gamma_1 \vdash t : T \dashv \Gamma_2$ for term typing. The latter shows that t may change the types contained in Γ_1 (for example, by assigning values to objects or by calling methods on them), giving rise to the final environment Γ_2. In the typing rules, we assume given the semantically defined judgement $\Delta \models p$.

We need a few auxiliary functions, defined in Figure 4, for looking up a class type, and for checking method overriding. We also use the following notation for interpreting paths in record types and type environments.

Definition 1 (Locations in record types).

- *We write* $(T + U)$ *to denote a record type* V *such that* $l : T' \in V$ *means either* $l : T' \in T$*, or* $l : T' \in U$ *when* $l : T' \notin T$.
- *If* $T \triangleq \{l_k : T_k{}^{k \in 1\ldots n}\}$*, then* $l_j : T_j \in T$ *and* $T.l_j \triangleq T_j$ *for any* $1 \le j \le n$.
- *If* $\mathsf{class}\ C : \Pi\Delta \lhd D\bar{x}.T\ \mathsf{is}\ M$ *and* $T.l = \Pi\Delta'.T'$*, then* $C.l = \Pi\Delta, \Delta'.T'$.

Definition 2 (Locations in environments).

- *If* $\Gamma = \Gamma_1, y : T_1, \Gamma_2$*, then* $\Gamma\{y \mapsto T_2\} \triangleq \Gamma_1, y : T_2, \Gamma_2$.
- *If* $\Gamma = \Gamma_1, y : \{l_k : T_k{}^{k \in 1\ldots n}\}, \Gamma_2$*, then* $\Gamma\{y.l_j \mapsto T'\} \triangleq \Gamma_1, y : \{l_k : T_k'{}^{k \in 1\ldots n}\}, \Gamma_2$ *where* $T_k' = T_k$ *for* $k \ne j$ *and* $T_j' = T'$ *for any* $1 \le j \le n$.

The rules for checking type formation (Figure 5) are standard for dependently-typed systems, except that they rely on the class definition rather than on a conventional system of kinds. In particular, Wf-App may seem unusual, since it uses the subtyping rules to check the kind of type T.

Subtyping rules are presented in Figure 6. Reflexivity and transitivity are explicitly expressed by rules. S-ClassL/R allow a nominal type C to be used in place of its structural type, obtained by function $\mathsf{ctype}(C)$ (Figure 4). S-Sub defines a relation between a class and its superclass as one where the class may define more index parameters than its superclass. In rules S-ΠL and S-ΣR, the index term i is "guessed" by the external constraint solver, closely following Dunfield's approach [5]. The β-reduction for types is included via rules S-β and S-App. The remaining rules are congruences, including S-.., contravariant on the argument's initial type (input) and covariant on the final one (output).

The typing rule for classes (Figure 7) checks the formation of components of each class. T-Meth uses the typing judgement for terms, checking conformance of types in the final environment against the defined post-types. Figure 8 presents the typing rules for terms. T-Var is used to access an identifier, either this or a parameter, while T-Field

$\boxed{\Delta;\Gamma_1 \vdash t : T \dashv \Gamma_2}$

$$\frac{\Delta \vdash \Gamma \qquad y : T \in \Gamma}{\Delta;\Gamma \vdash y : T \dashv \Gamma}\ (\textsc{T-Var}) \qquad \frac{\Delta;\Gamma \vdash y.l : T' \dashv \Gamma \qquad \Delta;\Gamma \vdash s : Ti \dashv \Gamma}{\Delta;\Gamma \vdash y.l := s : \mathsf{Empty} \dashv \Gamma\{y.l \mapsto Ti\}\{s \mapsto \mathsf{Empty}\}}\ (\textsc{T-AffAssign})$$

$$\frac{\Delta;\Gamma \vdash y : T \dashv \Gamma}{\Delta;\Gamma \vdash y.l : T.l \dashv \Gamma}\ (\textsc{T-Field}) \qquad \frac{\Delta;\Gamma \vdash y.l : C \dashv \Gamma \qquad \Delta;\Gamma \vdash s : C \dashv \Gamma}{\Delta;\Gamma \vdash y.l := s : \mathsf{Empty} \dashv \Gamma}\ (\textsc{T-UnAssign})$$

$$\frac{C.\mathsf{init} = \Pi\Delta'.T' \times \bar{U}..\bar{U}' \qquad \Delta \vdash \theta : \Delta' \qquad \Delta;\Gamma \vdash y.l, \bar{s} : T, \bar{U}\theta \dashv \Gamma}{\Delta;\Gamma \vdash y.l := (\mathsf{new}\ C)\bar{s} : \mathsf{Empty} \dashv \Gamma\{y.l \mapsto T'\theta\}\{\bar{s} \mapsto \bar{U}'\theta\}}\ (\textsc{T-NewAssign})$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash s_1 : T \dashv \Gamma \qquad T.l = \Pi\Delta'.(T_1..T_2) \times (U_1..U_2) \\ l \neq \mathsf{init} \qquad \Delta \vdash \theta : \Delta' \qquad \Delta \vdash T <: T_1\theta \qquad \Delta;\Gamma \vdash s_2 : U_1\theta \dashv \Gamma\end{array}}{\Delta;\Gamma \vdash (s_1.l)s_2 : \mathsf{Empty} \dashv \Gamma\{s_1 \mapsto T_2\theta\}\{s_2 \mapsto U_2\theta\}}\ (\textsc{T-Call})$$

$$\frac{\Delta;\Gamma_1 \vdash s_1, s_2 : T_1 \dashv \Gamma_1 \qquad \Delta;\Gamma_1 \vdash t_1, t_2 : T_2 \dashv \Gamma_2}{\Delta;\Gamma_1 \vdash \mathsf{if}\ s_1 = s_2\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 : T_2 \dashv \Gamma_2}\ (\textsc{T-If}) \qquad \frac{\Delta;\Gamma \vdash s_1, s_2 : T_1 \dashv \Gamma \qquad \Delta;\Gamma \vdash t : T_2 \dashv \Gamma}{\Delta;\Gamma \vdash \mathsf{while}\ s_1 = s_2\ \mathsf{do}\ t : \mathsf{Empty} \dashv \Gamma}\ (\textsc{T-While})$$

$$\frac{\Delta;\Gamma_1 \vdash t_1 : T_1 \dashv \Gamma_2 \qquad \Delta;\Gamma_2 \vdash t_2 : T_2 \dashv \Gamma_3}{\Delta;\Gamma_1 \vdash t_1; t_2 : T_2 \dashv \Gamma_3}\ (\textsc{T-Seq}) \qquad \frac{\Delta;\Gamma_1 \vdash t : T_2 \dashv \Gamma_2 \qquad \Delta \vdash T_2 <: T_1}{\Delta;\Gamma_1 \vdash t : T_1 \dashv \Gamma_2}\ (\textsc{T-Sub})$$

Figure 8: Rules for typing terms

is the rule for accessing an instance variable. A reference to an object of an index refined type acquires type Empty after appearing on the right-hand side of an assignment in T-AffAssign, thus implementing safe destructive updates (*cf.* T-UnAssign). The ownership discipline is also supported by T-Call which updates the final type environment with post-types, thus tracking the effects that the method produces on its arguments. Rules T-If and T-While use a simple reference equality.

Operational Semantics. Indices are simply an artifact of the type system; they do not exist on runtime. For this reason, rules in the operational semantics do not introduce additional complications.

6. CONCLUSION AND FUTURE WORK

In this paper, we have presented an object-oriented language with index refinement types, designed to support the verification of mutable objects. Our examples illustrate the main strengths and limitations of an approach that adopts a restricted form of dependent types as an approximation to decidable typechecking. We are currently implementing a prototype compiler for DOL, and studying the integration of richer index languages in domains of interest. We also expect to handle the aliasing concern in DOL by providing an alternative, less restrictive approach to affine types. To relax the notion of uniqueness, we intend to introduce an indirection into type environments in the style of [14].

7. REFERENCES

[1] L. Augustsson. Cayenne a language with dependent types. In *ICFP*, pages 239–250. ACM Press, 1998.

[2] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure and Interoperable Smart devices*, pages 49–69, 2005.

[3] E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, 2013.

[4] The Coq reference manual, version 8.4, 2012.

[5] J. Dunfield. *A Unified System of Type Refinements*. PhD thesis, Carnegie Mellon University, 2007. CMU-CS-07-129.

[6] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *TOPLAS*, 23(3):396–450, 2001.

[7] K. R. M. Leino. Extended static checking: A ten-year perspective. In *Informatics – 10 Years Back. 10 Years Ahead*, pages 157–175, 2001.

[8] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis-Napoli, 1984.

[9] C. McBride. Epigram: Practical programming with dependent types. In *Advanced Functional Programming*, volume 3622, pages 130–170. Springer, 2004.

[10] A. Nanevski, J. G. Morrisett, and L. Birkedal. Hoare type theory, polymorphism and separation. *Journal of Functional Programming*, 18(5-6):865–911, 2008.

[11] N. Nystrom, V. Saraswat, J. Palsberg, and C. Grothoff. Constrained types for object-oriented languages. In *OOPSLA*, pages 457–474. ACM, 2008.

[12] B. C. Pierce. *Types and programming languages*. MIT Press, 2002.

[13] B. C. Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2004.

[14] F. Smith, D. Walker, and J. G. Morrisett. Alias types. In *ESOP*, pages 366–381, 2000.

[15] H. Xi. Imperative programming with dependent types. In *LICS*, pages 375–387. IEEE Press, 2000.

[16] H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL*, pages 214–227. ACM Press, 1999.