

# Dependent Types for Class-based Mutable Objects

Joana Campos

LASIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal

jcampos@lasige.di.fc.ul.pt

 <https://orcid.org/0000-0002-2185-8175>

Vasco T. Vasconcelos

LASIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal

vv@di.fc.ul.pt

 <https://orcid.org/0000-0002-9539-8861>

---

## Abstract

---

We present an imperative object-oriented language featuring a dependent type system designed to support class-based programming and inheritance. Programmers implement classes in the usual imperative style, and may take advantage of a richer dependent type system to express class invariants and restrictions on how objects are allowed to change and be used as arguments to methods. By way of example, we implement insertion and deletion for binary search trees in an imperative style, and come up with types that ensure the binary search tree invariant. This is the first dependently-typed language with mutable objects that we know of to bring classes and index refinements into play, enabling types (classes) to be refined by indices drawn from some constraint domain. We give a declarative type system that supports objects whose types may change, despite being sound. We also give an algorithmic type system that provides a precise account of quantifier instantiation in a bidirectional style, and from which it is straightforward to read off an implementation. Moreover, all the examples in the paper have been run, compiled and executed in a fully functional prototype that includes a plugin for the Eclipse IDE.

**2012 ACM Subject Classification** Software and its engineering → Semantics

**Keywords and phrases** dependent types, index refinements, mutable objects, type systems

**Digital Object Identifier** 10.4230/LIPIcs.ECOOP.2018.13

**Supplement Material** ECOOP Artifact Evaluation approved artifact available at  
<http://dx.doi.org/10.4230/DARTS.4.3.1>

**Acknowledgements** LASIGE Research Unit, ref. UID/CEC/00408/2013

## 1 Introduction

Dependent types constrain types with values that specify intrinsic properties of programs. These sorts of types can represent concisely a number of invariants and prevent some classes of errors at compile time, rather than at runtime, which constitutes a step towards more reliable software. A key reason why dependent types are interesting is that they are a smooth extension of simple types. For example, using dependent types it becomes possible to express the non-negative balance  $b$  of a bank account as simply  $\text{Account}(b)$ , as well as ordered data structures, such as a binary search tree  $\text{BST}(l, u)$  whose elements can find a place within some minimum ( $l$ ) and maximum ( $u$ ) keys.

Previous work has shown how far one can go with dependent types in the context of logic and functional languages [2, 4, 18, 33, 53]. Agda [37], DML [51] and Idris [5] are noteworthy



## 13:2 Dependent Types for Class-based Mutable Objects

examples of functional languages that include advanced features and dependent types. Less work has been done in an imperative setting [9, 48], and none (that we know of) in class-based object-oriented programming with mutable objects.

While the benefits of a dependently-typed system in functional programming – increasing expressiveness and safety – are exactly the same for object-oriented programming, how to smoothly combine dependent types and mutable objects is still an open problem. For example: What object invariants to enforce with dependent types, and when to enforce them? How to track state in order to know which variables are modified and how? How to achieve the “safe substitutability principle” [31] with dependent types? The extra complexity that comes with developing the metatheory for a calculus that captures with dependent types the essential (but non-trivial) features of object orientation, such as mutable state and inheritance with subtyping, is challenging and requires a different approach.

In this paper, we formulate Dependent Object-oriented Language (DOL) as a smooth extension of simply typed Java-like languages that addresses the challenges of combining dependent types and object orientation. DOL offers a middle ground between traditional type systems and verification techniques, along the lines suggested by Leino [30] when discussing future challenges of the ESC static checker and the need to explore “more-than-types” systems to enforce program invariants. As argued, static verification is a powerful approach, despite being unsound, designed for “finding bugs in a program”, not for providing the guarantees of a type system, yet significantly more expressive than types. As it so happens, formal verification is not always suitable and is still too costly for mainstream adoption, often requiring prior training in logic or theorem proving. Without giving up soundness or falling back on dynamic checks, DOL is closer to existing programming methodologies, capturing via types a subset of the ESC properties.

We present a solution that allows programmers to start with standard types, writing code in an imperative style, and add more type information so as to gain additional guarantees. From simply typed, the type systems of mainstream object-oriented languages have already become more expressive and complex, namely when generics were introduced. DOL goes a step further introducing a restricted form of dependent types to enable special terms, called indices, to be parameters to classes and methods. By restricting the domain of the constraint language to that of linear inequalities over the integers, along the lines of DML [53], we render DOL’s type system decidable. The programmer simply needs to abstract the class declaration on properties they want to capture. For example, a class `Account` may be declared as follows: `class Account(b:natural){ balance: Integer(b)... }` where `natural` is a subset type that abbreviates `{x:integer | x ≥ 0}`. The index variable `b` is used to sharpen the type of fields and methods defined in `Account`, so that the typechecker can enforce through types a behaviour that forbids overdrafts. We say that `Account` defines a family of classes representing bank accounts whose instances can have many types, including the concrete type `Account<100>` obtained by instantiating `b` with `100`. Like generics, types in DOL support a variety of arguments. The difference is that in DOL the arguments to types are index terms that satisfy the specified constraints.

In certain states, some methods must not be available at the risk of violating object invariants. DOL provides support for the specification of method availability through fine-grained method signatures. For a `withdraw` method in the `Account` class, the signature should be roughly “`withdraw` takes an `Integer(m)` where  $0 \leq m \leq b$  on any `Account(b)` that becomes `Account(b - m)`”. The typechecker statically tracks objects and any state change, guaranteeing that calling `withdraw` with an invalid argument leads to a type error caught by the compiler. Moreover, to enforce behavioural subtyping, a subclass may reuse `Account` by declaring `extends Account(b)` which ensures that the invariant of the superclass is preserved in the subtype.

Given that objects may be mutable, aliasing in a language that allows types to change can result in a program “getting stuck”: if an `Account` object is aliased and its balance is changed by foreign code, reading from the alias will produce an unexpected result. In DOL, type varying objects are alias protected via a linear type discipline. A distinct category of type invariant, shared objects is allowed to coexist, but cannot subvert the linear system.

As a specification of typechecking, we give a declarative type system which extends with indices the Java notion of class types that take the form of  $C\bar{i}$ . From the dependent type theory, the system generalises simple function spaces to dependent function spaces  $\Pi a : I.T$  where the result type  $T$  can depend on the value of the argument  $a$ , restricted to special terms of index type  $I$ . Similarly, dependent sum types, written  $\Sigma a : I.T$ , generalise ordinary product types restricted to some constraint domain. The language also includes union types of the form  $T + U$  that eliminate the need of the unsafe `null` value. Moreover, the type system is able to record changes to mutable state. The calculus is a significant contribution of this paper, since it features the desirable property of type soundness, expressed via subject reduction and progress.

Then, we give an algorithmic type system which modifies the rules that require guessing quantifier instantiation [15, 16] and applies bidirectional typechecking [39] in order to distinguish rules that synthesize types from those that check terms against types already known. From this precise algorithm it is straightforward to read off an implementation.

We make the following contributions:

1. In contrast to other extensions to object-oriented programming, we define types that capture both the *immutable* and *mutable* state of objects (cf. [38]). A combination of index refinements and method signatures featuring input and output types enables a smooth integration of dependent types and class-based mutable as well as immutable, shared objects.
2. We let the type of an object change throughout the program based on a sound type system whereby a linear type discipline enforces unique references to type varying objects.
3. We provide support for single class inheritance as long as the subtype satisfies the index constraints defined by its supertype, and the inherited specifications remain meaningful in the context of the subclass.
4. We give a precise algorithm that is both sound and complete. The algorithm has an implementation in a prototype compiler for DOL that includes a plugin for the Eclipse IDE, a development tool that is widely used in the context of object-oriented languages but still new for dependently-typed languages.

## 2 DOL by Example

A class in DOL is declared just like any other class in a Java-like language, except that index variables may be introduced in the header and be used within the class to constrain member types. The class body contains fields and methods, including a constructor method named `init`. Like Java, DOL supports single class inheritance using the optional `extends` declaration. If omitted, the class is derived from the default superclass `Top`, a concrete class which has no fields or methods, except for the constructor.

### 2.1 Bank Account

Figure 1 defines the indexed class `Account` and its subclass `PlusAccount`. Notice that if we omit the extra type annotations in the example, we get plain Java-like code, with `Account` being simply a class type. However, when indexed, the class name `Account` denotes a family of

## 13:4 Dependent Types for Class-based Mutable Objects

```

1 class Account<b:natural> {
2   balance: Integer<b>
3
4   init(): Account<0> =
5     balance := 0
6
7   <m:natural>
8   [Account<b> ~> <b+m>]
9   deposit(amount: Integer<m>) =
10    balance := balance + amount
11
12  <m:natural{m≤b}>
13  [Account<b> ~> <b-m>]
14  withdraw(amount: Integer<m>) =
15    balance := balance - amount
16
17  getBalance(): Integer<b> =
18    balance
19 }
20 class PlusAccount<s,c,b:natural>
21   extends Account<b> {
22   savings: Integer<s>
23   checking: Integer<c>
24
25   init(): PlusAccount<0,0,0> =
26     balance, savings, checking := 0
27
28   <m:natural>
29   [PlusAccount<s,c,b> ~> <s+m,c,b+m>]
30   deposit(amount: Integer<m>) =
31     super.deposit(amount);
32     savings := savings + amount
33
34   <m:natural>
35   [PlusAccount<s,c,b> ~> <s,c+m,b+m>]
36   deposit2Checking(amount: Integer<m>) =
37     super.deposit(amount);
38     checking := checking + amount
39
40   <m:natural{m≤b ∧ b=s+c}>
41   [PlusAccount<s,c,b> ~>
42     <max(s-m,0),min(c,c-m+s),b-m>]
43   withdraw(amount: Integer<m>) =
44     super.withdraw(amount);
45     if amount ≤ savings {
46       savings := savings - amount
47     } else {
48       checking := checking - amount + savings;
49       savings := 0
50     }
51 }
```

 **Figure 1** The indexed class `Account` and its subclass `PlusAccount`.

classes. Instantiations, or concrete classes, represent bank accounts that cannot be overdrawn, and may have many types, namely `Account <0>`, `Account<1>`, ... where the occurrence of the index variable introduced in the class header is replaced by the corresponding value (an index term). State is somehow exposed in types through indices, but fields are always *private* to a class, even if we do not use the corresponding keyword.

The special `init` method behaves as a typical constructor that initialises fields, creating a fresh object assigned the proper (or concrete) type `Account<0>` (line 4). One consequence of objects having different types is that the compiler must track state changes throughout the program, namely when client code creates an account object and calls methods on it:

```

acc := new Account(); // acc: Account<0>
acc.deposit(100);    // acc: Account<100>
acc.withdraw(30)    // acc: Account<70>
```

**State Modifying Methods.** We give indexed signatures to methods that are defined in indexed classes. For example, the `withdraw` method (lines 12–15) must be invoked on a receiver of type `Account<b>`, accepts an amount of type `Integer<m>`, modifies the type of the receiver from the initial `Account<b>` to the final `Account<b-m>`, and does so for any amount `m` that is a natural number smaller or equal to the balance. In the formal language, the method type, written  $\Pi m : \{x: \text{integer} \mid 0 \leq x \leq b\}.T$ , is a universal type that binds the index variable `m` in a type `T` (where `T` represents the types of the implicit and explicit parameters and the return type from the example), so that one can mention `m` in `T`. The scope of the index variable `m` is therefore local; it may appear in the method signature, but not outside. The type `[Account<b> ~> <b-m>]`, read “`Account<b>` becomes `Account<b-m>`”, is an abbreviation for a pair of types. The first type is seen as the input type of the (implicit) receiver and the second

one is viewed as its output type. Finally, when a method does not explicitly declare a return type, the typechecker assumes the supertype `Top`.

To illustrate the precision of the types in DOL, here is a variant of the preceding example, changed by adding a second call to method `withdraw` that violates the object invariant:

```
acc := new Account(); // acc: Account<0>
acc.deposit(100); // acc: Account<100>
acc.withdraw(70); // acc: Account<30>
acc.withdraw(50) // Type error: 50 > 30
```

Both `deposit` and `withdraw` are examples of methods that change state, which we sometimes call *type varying* methods, having to explicitly declare the input and output types of their implicit receivers. On the contrary, in *type invariant* methods, that is, methods whose input and output types coincide, receiver types may be omitted. The `getBalance` (lines 17–18) method provides one such example.

**Base Types and Literals.** Constants and operators are used in the programmer’s language only to make arithmetic and logic operations look more familiar, since they are not part of DOL’s core language. In fact, constants and operators are desugared into object references and method calls. Formally, `Integer` and `Boolean`, implemented natively, are families of classes. For example, the `Integer` “interface” includes the following types:

```
class Integer<i:integer> {
    init(): Integer<0>
    ⟨j:integer⟩ + (value: Integer⟨j⟩): Integer⟨i+j⟩
    ⟨j:integer⟩ ≤ (value: Integer⟨j⟩): Boolean⟨i ≤ j⟩
    ...
}
```

Each desugared object of a primitive class is assigned a singleton type, with the constants used in the examples representing the values on which the types depend. Technically, the argument `100` from an earlier example is an object reference of type `Integer<100>`, obtained by creating a new location, and subtraction is translated into the call `balance.minus(amount)` before typechecking.

**Controlled Aliasing.** Aliasing is part of what makes mutable objects useful in programming. However, shared state can be tricky to handle in a type system such as that of DOL, where the type of a variable may no longer be a fixed class type; instead, it may be a (dependent) type that changes throughout the program. In DOL, the potential sources of aliasing problems are assignment and parameter passing. We adopt a solution that uses linear control of those objects defined by type varying classes. We say that a class is type varying when at least one of its methods is type varying, giving different input and output types to its receiver. Because the `Account` class is type varying as per methods `deposit` and `withdraw`, the type system forbids creating aliases of instances of `Account`. Here is how DOL’s typechecker handles aliasing:

```
alias := acc; // alias: Account<30>
acc.withdraw(20); // Type error: acc has been consumed!
alias.withdraw(10) // alias: Account<20>
```

Instead of creating an alias, the assignment “consumes” variable `acc`, removing it from the typing context; hence, the call to `withdraw` in the second line is forbidden, with `alias` being the only variable available in the typing context.

Similarly, our type system ensures that parameters are used correctly by treating these references linearly when needed. To show that our language can be flexible, despite the linear

## 13:6 Dependent Types for Class-based Mutable Objects

restriction, we could add a `transferTo` method to debit some amount from the current account and credit into another account given as parameter, using the following implementation:

```
(a:natural,m:natural{m≤b})
[Account⟨b⟩ ↪ ⟨b-m⟩]
transferTo(other: Account⟨a⟩, amount: Integer⟨m⟩): Account⟨a+m⟩ =
    var local := other; // other has been consumed!
    withdraw(amount);
    local.deposit(amount);
    local
```

On the other hand, we say that a class is type invariant when its methods are type invariant, i.e. when the input and output types coincide (or are omitted) in all methods. The native `Integer` and `Boolean` provide two examples of such classes whose objects can be freely shared. Since each new assignment creates a new location, instances of these classes carry their types unchanged irrespective of being accessed or aliased.

**Inheritance and Subtyping.** Adapted from JML [13, 29], the `PlusAccount` class illustrates how DOL can achieve the “safe substitutability principle” [31] via indexed types. By declaring `extends Account⟨b⟩`, we make the subtype inherit the `Account`’s only field, as well as all of its methods (except the constructor). In `PlusAccount`, we declare two extra index variables, `s` and `c`, and use them to constrain fields `savings` and `checking` that hold two portions of the `balance`.

We can think of `PlusAccount` as extending the behaviour of `Account` by providing additional fields and methods. So, the `deposit` method directly inherited (if not overridden) from `Account` will be given the following type:

```
(m:natural)
[PlusAccount⟨s,c,b⟩ ↪ ⟨s,c,b+m⟩]
deposit(amount: Integer⟨m⟩) = ...
```

However, we want relate the two new fields in the subclass with the superclass’s field by enforcing that  $b=s+c$  via method signatures. We override the `deposit` method (lines 28–32) that adds the amount both to the account’s balance, by calling the superclass method, and the `savings` field. A new method `deposit2Checking` (lines 34–38) also adds the given amount to the `checking` field. The `withdraw` method (lines 40–50) must be redefined in order to take out the amount from each balance portion. DOL’s typechecker gives the index equations issued by types to an external constraint solver, and asks if they hold.

Common mistakes that violate the (inherited) invariant are readily detected. For example,

```
(m:natural{m≤s})
[PlusAccount⟨s,c,b⟩ ↪ ⟨max(s-m,0),min(c,c-m+s),b-m⟩]
withdraw(amount: Integer⟨m⟩) = ...
```

yields a type error, since a subtype cannot accept a *stronger* requirement, that is, it cannot accept less arguments as valid [31] (it should be clear that the constraint  $m \leq s$  does not imply  $m \leq b$  under the assumption that  $b=s+c$ ). A subtler type error is found in the following variant:

```
(m:natural{m≤b})
[PlusAccount⟨s,c,b⟩ ↪ ⟨max(s-m,0),min(c,c-m+s),b-m⟩]
withdraw(amount: Integer⟨m⟩) = ...
```

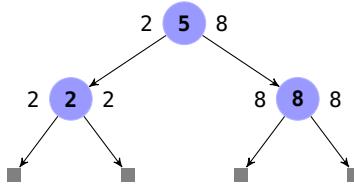
The problem here is that the constraint  $m \leq b$  does not relate the value of `amount` with the two portions of the balance, unlike the indices in the output type. Specifically, the index refinement does not provide enough evidence that allows the typechecker to conclude, after interaction with the solver, that the index term  $\min(c, c - m + s)$  is a natural number that can safely replace the index variable `c` introduced in the class header.

```

1 class BST<l,u:integer> {
2   root: Nil + Node<l,k,u>
3     where k:integer{l ≤ k ≤ u}
4
5   init(): BST<2,1> =
6     root := new Nil()
7
8   <v:integermin(l,v),max(u,v)>]
10  insert(value: Integer<v>) = ...
11
12  <v:integerclass Nil {
17   init(): Nil = skip
18 }
19
20 class Node<l,k,u:integer{l ≤ k ≤ u} {
21   key: Integer<k> // fields
22   left: Nil + Node<l,k1,u1>
23     where k1,u1:integer{l ≤ k1 ≤ u1 ≤ k}
24   right: Nil + Node<l1,k1,u>
25     where l1,k1:integer{k ≤ l1 ≤ k1 ≤ u}
26
27   <v:integernew Nil(), new Nil()
31
32   <v:integermin(l,v),k,max(u,v)>]
34   add(value: Integer<v>) = ...
35
36   <v:integerwhere k1:integer{l ≤ k1 ≤ u}]
39   deleteChild(value: Integer<v>) = ...
40 }

```

**Figure 2** Classes that implement a dependently-typed binary search tree.



**Figure 3** The diagrammatic representation of an object of type `Node<2,5,8>` where labels at each tree node denote the smallest and greatest keys appearing in the tree.

## 2.2 Binary Search Tree

Binary search trees can naturally be described by the discipline of dependent types [14, 28, 34, 47]: a binary search tree is either empty or nonempty in which case it has two subtrees that are binary search trees, and the key in the root node of the binary search tree is greater than all the keys appearing in its left subtree and smaller than all the keys appearing in its right subtree. This example shows that our type system can be precise and expressive while implementations remain as usual. The effort of programming in DOL is essentially to come up with the right type.

We implement the binary search tree in an imperative style, allowing subtrees to be modified *in place*. In Figure 2, we show the types, defining `BST` as the “public” family of classes that creates and manages both empty and nonempty trees using `Nil` (a proper class) and `Node` (a family of classes). Our binary search tree contains integer numbers included in the *loose* pair of bounds `<l,u:&bignum>` in the header of `BST` that can be used to define an *interval* `[l,u]`. Any element in a tree will find a place within the minimum (`l`) and maximum (`u`) keys.

While many approaches have been proposed [6, 8, 17] to handle Hoare’s billion dollar mistake [23], DOL provides an elegant solution using union types (cf. [24]) that enable programmers to build imperative linked data structures in a null-free style – object references are, after all, the only values in DOL. Union types (denoted by `T + U`) represent objects that can be of any of the specified types. Note that the lack of `null` in DOL means that every variable must be initialised, and that every variable of a union type must be analysed by way of a `case` construct before being used.

## 13:8 Dependent Types for Class-based Mutable Objects

So, class `BST` has a single field `root` that is either `Nil` or `Node`. A dependent existential quantified constructor (denoted by `where ...`) is used to keep track of the key, hidden in the field's type, so that the binary search tree invariant can be maintained. We write it as a dependent sum type in the formal language of the form  $\Sigma k : \{x: \text{integer} \mid l \leq x \leq u\}.\text{Node } l \ k \ u$ . Notice that this type conforms to the constraint ( $l \leq k \leq u$ ) issued by the signature of class `Node` that ensures the binary search order invariant.

The special `init` method (lines 5–6) creates an empty binary search tree to which we give the type `BST<2,1>`, making `root` an instance of `Nil`. When inserting a value in an empty tree, we replace an instance of `Nil` with an instance of `Node` with no children (a leaf). Then, when inserting in a nonempty tree, we recursively push the requirements of data inward, requiring that the value at each node falls within the interval  $[l,u]$ . For example, a type `BST<2,8>` may represent the binary search tree whose `root` of type `Node<2,5,8>` (depicted in Figure 3) issues the minimum and maximum keys outward. The value 5 stored in the root is not exposed, but is internally constrained by the tree bounds.

The `Node` class defines a field `key`, which holds the node value, and fields `left` and `right` that may represent the two subtrees. We use union and existential types, again pushing the data requirements inward to the types of the `left` and `right` fields. For example, the type of the `left` field (lines 22–23) enforces the fact that all the values appearing in the left subtree must be in the interval defined by  $[l,k]$ , so that the binary search tree invariant can be maintained. Similarly, the type of the `right` field enforces the fact that all the values appearing in this subtree must be included in  $[k,u]$ . The `init` method (lines 27–30) creates a leaf by accepting an integer value to be stored in the `key` field, making both `left` and `right` instances of `Nil`. By definition, leaf nodes are such that  $l=k=u$ . So, for example, `Node<2,2,2>` may be the type of the left subtree (a leaf) in Figure 3.

**BST Insertion.** We now define the `insert` method in the `BST` class, which takes as argument an integer `value` and provides a useful demonstration of a case discrimination construct:

```
<v:integer>
[BST<l,u> ~> <min(l,v),max(u,v)>]
insert(value: Integer<v>) =
  case root {
    Nil => root := new Node(value)
    Node => root.add(value)
  }
```

The `Node` class implements the main insertion algorithm. Its method `add` takes an argument similar to the one above, and also uses a case discrimination construct:

```
<v:integer>
[Node<l,k,u> ~> <min(l,v),k,max(u,v)>]
add(value: Integer<v>) =
  if value < key {
    case left {
      Nil => left := new Node(value)
      Node => left.add(value)
    }
  } else if value > key {
    case right {
      Nil => right := new Node(value)
      Node => right.add(value)
    }
  }
```

We add an element to the tree by comparing the `value` to the `key` stored at each node and recursively descending into the appropriate subtree until a leaf is reached that allows adding the new node.

The precise types given to the `BST` and `Node` classes allow the typechecker to detect a number of common programming errors. For example, the compiler will report a type error if we try to call the `add` method as follows:

```
(v:integer)
[BST(l,u) ~> (min(l,v),max(u,v))]
insert(value: Integer(v)) =
  root.add(value)
```

Because the `root` field declares a union type, we cannot call a method directly on it; first, we must use a `case` construct to analyse its type and discover whether the object is an instance of `Nil` or `Node`, noting that, at each branch, the typechecker requires that `root` be bound to only one of the types which are subtypes of the union of types. This guarantees that either branch is taken and its execution succeeds.

Similarly, the compiler will object to the wrong conditional test below:

```
(v:integer)
[Node(l,k,u) ~> (min(l,v),k,max(u,v))]
add(value: Integer(v)) =
  if value > key {
    case left {
      Nil => left := new Node(value)
      ...
    }
  }
```

Here, the compiler will report inconsistent constraints. The `case` construct is correctly used to find out that `left` is a `Nil`. Then, the assignment changes the type of the `left` field to `Node(v,v,v)` (which is the type given to it by `init`). However, the compiler assumes  $v > k$  from the conditional test, after which will not be able to assert  $v \leq k$  issued from the new type of the `left` field. Recall the constraints on the declared type of `left`, requiring its value be left-bounded by the minimum key  $l$  (which has become  $v$ ) and right-bounded by value  $k$  (known to be also  $v$ ) such that  $l \leq k$ . Again, DOL relies on the external constraint solver to statically verify that the specified constraints hold.

**BST Deletion.** Deletion from the binary search tree may involve removing a key not only from the tree's leaf nodes but also from an interior node, which requires some sort of rearrangement of the tree structure. Moreover, unlike insertion, in which `min` and `max` could be used to issue the new value's standing vis-à-vis the minimum and maximum keys existing in the tree, deletion delivers the same binary search tree where the new minimum or maximum may be hidden in the subtrees.

However, we can still ensure via types that the tree after deletion is within the bounds, no matter where the key removal occurs (from a fringe or the middle of the tree). The `remove` method is implemented as usual:

```
(v:integer)
remove(value: Integer(v)) =
  case root {
    Nil => skip
    Node =>
      if root.isLeaf(value) {
        root := new Nil()
      } else {
        root.deleteChild(value)
      }
  }
```

$P ::= \bar{L}$	(programs)
$L ::= \text{class } C : \Delta \text{ extends } T\{\bar{l} : \bar{T}\} \text{ is } \{\bar{M}\}$	(classes)
$M ::= m(x) = t$	(methods)
$T ::= \bar{C}i \mid \Pi a : I.T \mid \Sigma a : I.T \mid T + T$ $\mid T \times T \mid T \rightsquigarrow T \mid T \rightarrow T$	(types)
$t ::= x \mid f \mid \text{new } C() \mid f := t \mid t; t \mid m(t)$ $\mid f.m(t) \mid \text{case } f \text{ of } (C_k \Rightarrow t_k)_{k \in 1,2}$ $\mid \text{if } t \text{ then } t \text{ else } t \mid \text{while } t \text{ do } t$	(terms)
$\Delta ::= \epsilon \mid \Delta, a : I$	(index contexts)
$I ::= \text{integer} \mid \text{boolean} \mid \{a : I \mid p\}$	(index types)
$i ::= a \mid n \mid i \oplus i \mid p$	(index terms)
$p ::= \text{false} \mid \text{true} \mid \neg p \mid i \oslash i \mid p \oslash p$	(propositions)
$\oplus ::= + \mid -$	(arithmetic operators)
$\oslash ::= < \mid \leq \mid \doteq \mid \geq \mid >$	(relational operators)
$\oslash ::= \wedge \mid \vee$	(logical operators)

■ **Figure 4** Top-level syntax.

## 3 The DOL Language

The core language is a desugared version comprising all the properties informally described in the examples. It builds on the core sequential language of Gay et al. [21], which allows us to simplify proofs while keeping them manageable. We adapt and extend that language in three ways. (1) We replace session types with dependent types and study the consequences of this idea. (2) We incorporate inheritance and nominal subtyping, a feature absent from the base language. (3) We combine linear and unrestricted objects in the formalisation, building a less restrictive type system than the original one.

### 3.1 Syntax

Following standard practice [21], the formal language omits some features of the practical syntax used in the examples, so as to simplify the proofs, even though our prototype includes them. Below, we summarize the main differences.

- Primitive values as used in the examples are translated into object references, which are the only values in our language, and all computations are performed by calling methods. This lightens the type system without affecting expressivity.
- All methods have exactly one parameter. A method written  $m() = t$  abbreviates  $m(\text{top}) = t$  where  $\text{top}$  of type  $\text{Top}$  is used as a dummy parameter. Defining methods that take an arbitrary number of parameters does not introduce any major technical challenge.
- Local variables are omitted, since they can be simulated by a parameter or extra fields.

We define the top-level syntax in Figure 4. Identifiers are drawn from the following disjoint countable sets: that of class names (denoted by  $B, C, D$ ), that of fields (denoted by  $f, g$ ), that of methods (denoted by  $m$ ), that of object variables (denoted by  $x, y, o$ ), and

that of index variables (denoted by  $a, b$ ). Labels  $l$  identify class members, that can either be fields or methods. The metavariables  $T, U, V, W$  range over object types;  $I, J$  range over index types; and  $i, j$  range over index terms.

Programs  $P$  consist of collections of class declarations  $L$ . A class family, written  $\text{class } C : \Delta \text{ extends } T\{\bar{l} : \bar{T}\} \text{ is } \{\bar{M}\}$ , associates a class named  $C$  to an index context  $\Delta$ , a supertype  $T$ , a sequence of member declarations  $\bar{l} : \bar{T}$  (field and method signatures), and a sequence of method implementations  $\bar{M}$ . An index context maps index variables to index types, fixing the class family arity, with each entry having the form  $a : I$ . A concrete or proper type, written  $C\bar{i}$ , is obtained by instantiating a class family with indices in application position. Index variables in  $\Delta$  can be used to constrain types inside the class, including that of the explicit superclass  $T$ , where  $T$  is of the form  $D\bar{i}$ . (As we will see later, this restriction is enforced by the typing rules.) Finally, a method is implemented separately from its signature as  $m(x) = t$ , where  $t$  is the method body and, for simplicity,  $x$  its single parameter.

**Types.** Types  $T$  either classify objects or build method signatures. They can be of the following seven forms:

- A type  $C\bar{i}$  extends with indices the Java notion of class types.
- A universal dependent type constructor, written  $\Pi a : I.T$ , where  $a$  may occur free in  $T$ , is a type that maps elements of the index type  $I$  to elements in the type  $T$ . It is used to build method signatures.
- An existential type constructor, written  $\Sigma a : I.T$ , where  $a$  may occur free in  $T$ , also maps elements of the index type  $I$  to elements in the type  $T$ , with the index variable  $a$  representing some unknown value in  $T$ . It is used to represent undetermined properties of a concrete object type.
- A union type  $T + T$  classifies the set of objects belonging either to the left or the right type. It is used to define a supertype grouping independently developed classes.
- A product type, written  $T \times T$ , is used in method signatures, with the first type classifying the current object `this`, implicitly passed to the method, and the second one classifying the only explicit parameter.
- A parameter type of the form  $T \rightsquigarrow T$  relates the two components that classify the current object `this`, the input type and a possibly different output type.
- A method type, written  $T \rightarrow T$ , maps the type of the parameters to a return type.

**Terms.** Terms  $t$  are fairly standard, except for some restricted forms that allow the type system to record more precisely how the types of objects vary. The variable  $x$  denotes a parameter. There is no qualified  $x.f$ . Instead, field access, written  $f$ , is only defined for a shared field (a restriction enforced by the typing rules), or in combination with assignment, method calls and case constructs. This is part of the linear control of objects. All fields are private in the sense that every  $f$  always refers to a field of the current object (cf. [21]). Object creation (`new C()`) does not take any parameters. Assignment ( $f := t$ ) is defined in terms of a non-standard swap operation in the style of [21]. The operation assigns the value of  $t$  to the field  $f$  and returns the old value of  $f$  as its result. This prevents aliasing linear fields in terms such as  $f_2 := (f_1 := t)$ . The sequential term composition  $(t; t)$  is standard. Method call is available both on the current object itself (a *self call*), written  $m(t)$ , and on a field of the current object `this`, written  $f.m(t)$ , but not on a parameter or an arbitrary term for that matter. This is because calling a method may change the type of the object on which the method is called. Note that the type system only records changes on the type of

$T ::= \dots \mid C[F]$	(types)
$F ::= \{\bar{f} : \bar{T}\}$	(field types)
$r ::= o \mid r.f$	(paths)
$t ::= \dots \mid \text{return } t$	(terms)
$\theta ::= \epsilon \mid \theta, i/a$	(index substitutions)
$\Delta ::= \dots \mid \Delta, p$	(index contexts)
$\Gamma ::= \epsilon \mid \Gamma, x : T$	(object contexts)
$K ::= \star \mid \Pi a : I.K$	(kinds)
$h ::= \epsilon \mid h, o = R$	(heaps)
$R ::= C\{\bar{f} = \bar{o}\}$	(object records)
$S ::= (h * r, t)$	(states)
$\mathcal{E} ::= [] \mid f := \mathcal{E} \mid \mathcal{E}; t \mid m(\mathcal{E}) \mid f.m(\mathcal{E})$	(evaluation contexts)
$\mid \text{return } \mathcal{E} \mid \text{if } \mathcal{E} \text{ then } t \text{ else } t \mid \text{while } \mathcal{E} \text{ do } t$	

Figure 5 Extended syntax, used only in the type system and operational semantics.

the current object `this`, the only one that can access its fields. To simplify, the `case` construct may only depend on a field, taking the form of `case f of (Ck ⇒ tk)k∈1,2` where *f* plays the role of the binding occurrence in the branches. Conditionals and while loops are standard.

**Index Refinements.** Index types *I* comprise the integer and boolean types, as well as the subset type of the form  $\{a : I \mid p\}$ . Index terms *i* include some of the possible index constructs, namely variables, integer literals, arithmetic operations, and also propositions, which take the form of the truth values, the negation and linear inequalities. We omit functions `max` and `min` from the examples as they do not introduce any additional technical challenge.

### 3.2 Additional Syntax Not Available to Programmers

Figure 5 defines syntactic extensions required for the formal system only. The internal type  $C[F]$  (cf. [21]) is an alternative form of an object type that contains the class name *C* and a record field typing *F* that provides types for all the fields of *C*, including the inherited ones. For example,  $C[\{f_1 : T_1, f_2 : T_2\}]$  is the internal type of an object of *C* having two fields of types *T<sub>1</sub>* and *T<sub>2</sub>*, which may be defined either in *C* or in any of its superclasses. The internal type, used to classify the current object (`this`) in the context, cannot be the type of an arbitrary term, which never evaluates to `this` (as enforced by the typing rules). Instead, the purpose of the internal type is to allow the current object (`this`) to access its own fields, for typechecking assignment, method calls and case constructs, operations that may change its type through the field typing.

Terms evaluate to object references *o*, the only values in our language. To simplify, we do not define a separate syntactic category for object references. Instead, object references *o* are a subset of the variable names. Paths *r* in the style of [21] represent locations in the heap, formed by the top-level object followed by a sequence of an arbitrary number of fields. For example, if *r* indicates the path of the currently active object, when a method call on a

field  $f$  relative to  $r$  is entered,  $r.f$  becomes the path that indicates the new current object that becomes active. The `return` term represents an ongoing method call during which the path changes as described above. Paths and the `return` term are constructs belonging to the operational semantics.

Substitutions  $\theta$  map index variables to index terms. Index contexts  $\Delta$  are extended to accept propositions  $p$ . Object contexts  $\Gamma$  map object variables to types.

Types are classified into kinds  $K$ , much the same way as terms are classified into types. Kind  $\star$  characterizes proper types, while kind  $\Pi a : I.K$  classifies families of classes, i.e. types that have to be applied to index terms to form proper types. We only ever need to check for proper types (with kind  $\star$ ). In fact, the only way to construct a type of a kind other than  $\star$  is by declaring an indexed class. So, in the bank account example, the class family `Account` has kind  $\Pi b : \text{natural}.\star$ , and an instantiation, say `Account 0`, of kind  $\star$ , denotes the proper type of an object reference.

A heap  $h$  is a mapping from object references  $o$  to object records  $R$ . We assume three special objects (`top` : `Top`, `false` : Boolean `false`, `true` : Boolean `true`) that are initially placed in the heap. The heap produced by the operation  $h, (o = R)$  contains a new mapping from object reference  $o$  to record  $R$ . The operation of adding this binding to the heap  $h$  is only defined if  $o \notin \text{dom}(h)$ . Note that the order in  $h$  is irrelevant. Records  $R$  are instances of classes, represented by  $C\{\bar{f} = \bar{o}\}$ , comprising the class of the object followed by a mutable record mapping field names to object references.

The operational semantics is defined as a reduction relation on states  $S$  of the form  $(h * r, t)$ , consisting of a heap  $h$ , a path  $r$  that represents the current object, and a term  $t$ . Evaluation contexts  $\mathcal{E}$  are defined in the style of [46]. Intuitively, an evaluation context is a term with a hole  $\underline{\phantom{x}}$  at the point where the next reduction step must take place in a call-by-value evaluation order;  $\mathcal{E}[t]$  is the term obtained by replacing the hole in  $\mathcal{E}$  by term  $t$ .

### 3.3 Static Semantics

We typecheck our language with respect to one index context  $\Delta$ , and one object context  $\Gamma$ . The ordering is important in the index context, because of (index) variable-to-type dependencies, and irrelevant in the object context. For example, an index context  $(\Delta_1, a : I, \Delta_2)$  is said to be well-formed if  $a \notin \text{dom}(\Delta_1) \cup \text{dom}(\Delta_2)$  and  $a \notin \text{FV}(\Delta_1)$ ; an index context such as  $(c : \{b : \text{integer} \mid b \geq a\}, a : I)$  is ill-formed. We give the subtyping and typing rules for top-level terms in the sections that follow; we omit rules for the index language, kinding, context formation and typing. First, we give an overview of index refinements and substitution.

**Index Refinements and Substitution.** Our formulation of index refinements requires a way to somehow decide the semantically defined relation  $\Delta \models p$  in the style of Xi and Pfenning [47, 52, 53]. The binding occurrences of index variables appear in subset types, and also in types and kinds in the object language. We say that  $a$  occurs bound in  $p$  within  $\{a : I \mid p\}$ , in  $T$  within  $\Pi a : I.T$  and  $\Sigma a : I.T$ , and in  $K$  within  $\Pi a : I.K$ .

To simplify the proofs, and to avoid having to rename bound variables in substitution, we follow Barendregt's variable convention [3] whereby the names of bound variables must all be distinct from each other and from any other variables occurring free in terms and types.

We denote by  $i_1[i_2/a]$  the capture-avoiding substitution of  $i_2$  for the free occurrences of  $a$  in  $i_1$ . Index substitutions are defined inductively on the structure of index terms. For example,  $(i_1 + i_2)[i_3/a]$  is defined as  $i_1[i_3/a] + i_2[i_3/a]$ . A single index substitution is extended pointwise to multiple index substitution  $\theta$ , which maps index variables to index terms, by defining  $i\epsilon \triangleq i$  and  $i_1([i_2/a], \theta) \triangleq (i_1[i_2/a])[\theta]$ .

$\boxed{\Delta \vdash T <: U}$  Under context  $\Delta$ , type  $T$  is a subtype of  $U$

$$\begin{array}{c}
 \frac{\text{class } C : (\bar{a} : \bar{I}) \text{ extends } T\{\_\} \text{ is } \{\_\} \quad \Delta \vdash \bar{i} : \bar{I} \quad \Delta \vdash T[\bar{i}/\bar{a}] : *}{\Delta \vdash Ci <: T[\bar{i}/\bar{a}]} \text{ (S-SUPER)} \\
 \frac{\Delta \models \bar{i} \doteq \bar{j} \quad \Delta \vdash C\bar{j} : *} {\Delta \vdash C\bar{i} <: C\bar{j}} \text{ (S-APP)} \quad \frac{\Delta \vdash T[i/a] <: U \quad \Delta \vdash i : I}{\Delta \vdash \Pi a : I.T <: U} \text{ (S-IIL)} \\
 \frac{\Delta, a : I \vdash T <: U}{\Delta \vdash T <: \Pi a : I.U} \text{ (S-IIR)} \quad \frac{\Delta, a : I \vdash T <: U}{\Delta \vdash \Sigma a : I.T <: U} \text{ (S-SL)} \\
 \frac{\Delta \vdash T <: U[i/a] \quad \Delta \vdash i : I}{\Delta \vdash T <: \Sigma a : I.U} \text{ (S-SR)} \quad \frac{\Delta \vdash T_1 <: U \quad \Delta \vdash T_2 <: U}{\Delta \vdash (T_1 + T_2) <: U} \text{ (S-+L)} \\
 \frac{\Delta \vdash T <: U_k}{\Delta \vdash T <: (U_1 + U_2)} \text{ (S-+R}_k\text{)} \quad \frac{\Delta \vdash T_1 <: U_1 \quad \Delta \vdash T_2 <: U_2}{\Delta \vdash (T_1 \times T_2) <: (U_1 \times U_2)} \text{ (S-}\times\text{)} \\
 \frac{\Delta \vdash \bar{T} <: \bar{U}}{\Delta \vdash C[\{\bar{f} : \bar{T}\}] <: C[\{\bar{f} : \bar{U}\}]} \text{ (S-RECORD)} \quad \frac{\Delta \vdash T_1 <: T_2 \quad \Delta \vdash T_2 <: T_3}{\Delta \vdash T_1 <: T_3} \text{ (S-TRANS)}
 \end{array}$$

Figure 6 Subtyping rules.

The judgement for deriving  $\theta$  is of the form  $\Delta_1 \vdash \theta : \Delta_2$  where, under the assumptions in context  $\Delta_1$ , we think of  $\Delta_2$  as the input and  $\theta$  as the output. The rules require that  $\Delta_2$  and  $\theta$  have the same arity and that each substituent is well-formed in the context. Specifically, for each substitution  $i/a$ , there is an entry  $a : I$  such that  $\Delta_1 \vdash i : I$ . As for index terms, application of a substitution  $\theta$  to a type  $T$ , denoted by  $T[\theta]$ , is standard, defined inductively on the structure of  $T$ .

### 3.3.1 Subtyping

Term typing and method overriding rely on the subtyping relation defined as the reflexive and transitive closure of the inheritance relation as in Java, guided by the “safe substitutability principle” [31]. The judgement  $\Delta \vdash T <: U$  asserts that  $T$  is a subtype of  $U$  under the assumptions in context  $\Delta$ . We give the rules for subtyping in Figure 6.

All types in the top-level language are subject to subtyping, except for  $\rightsquigarrow$  and  $\rightarrow$ , since these types cannot arise from terms, and are not used to check method overriding (cf. Figures 9 and 10). The internal field typing is also subject to subtyping in order to check compatibility between fields of the same class. This relation is always derived with respect to the internal type of the current object (`this`), the only one that has access to its own fields.

S-SUPER is completely standard for object-oriented languages, adjusted to dependent types. Because class `Top` does not declare a supertype, it follows that `Top` is a supertype of every other type. By S-APP, subtyping is reflexive on class types, extended pointwise to all possible applications of the class type that satisfy the  $\models$  relation, and by rule S-TRANS, subtyping is transitive.

Regarding S-IIL and S-IIR, the left rule instantiates the index variable  $a$  to  $i$  in the subtype, while the right rule relates two types  $T$  and  $U$  provided the variable  $a$  does not appear free in  $T$ . The reasoning for S-SL and S-SR is similar, yet inverted. Following Barendregt’s variable convention [3], we implicitly assume that the variable  $a$  in the extended context of both S-IIR and S-SL is distinct from all the variables already in  $\Delta$ .

$\text{classof}(T) = C$	$\text{classof}(C\bar{i}) = C$	$\text{classof}(\Sigma a : I.T) = \text{classof}(T)$
$\text{fields}(T) = U$	$\text{fields}(\text{Top}) = \text{Top}[\{\}]$	$\text{fields}(\Sigma a : I.T) = \Sigma a : I.\text{fields}(T)$
$\text{class } C : (\bar{a} : \bar{I}) \text{ extends } D\bar{j}\{\bar{f} : \bar{U}, \bar{m} : \_\} \text{ is } \{\_\}$	$\text{fields}(D\bar{j}[\bar{i}/\bar{a}]) = D[\{\bar{g} : \bar{V}\}]$	$\text{fields}(C\bar{i}) = C[\{\bar{g} : \bar{V}\} \sqcup \{\bar{f} : \bar{U}[\bar{i}/\bar{a}]\}]$
$\text{mtype}(m, C\bar{i}) = T$		
	$\frac{\text{class } C : (\bar{a} : \_) \text{ extends } \{\dots, m : U, \dots\} \text{ is } \{\_\}}{\text{mtype}(m, C\bar{i}) = U[\bar{i}/\bar{a}]} \text{ (MT-CLASS)}$	
	$\frac{\text{class } C : (\bar{a} : \_) \text{ extends } D\bar{j}\{\bar{l} : \_\} \text{ is } \{\_\} \quad m \notin \bar{l}}{\text{mtype}(m, C\bar{i}') = \text{mtype}(m, D\bar{j}[\bar{i}'/\bar{a}])[C\bar{i}/D]} \text{ (MT-SUPER)}$	
$\text{mbody}(m, C) = \lambda x.t$		
	$\frac{\text{class } C : \_ \text{ extends } \{\_\} \text{ is } \{\dots, \text{init}() = \bar{f} := \text{new } \bar{C}(), \dots\}}{\text{mbody}(\text{init}, C) = \bar{f} := \text{new } \bar{C}()} \text{ (MB-INIT)}$	
	$\frac{\text{class } C : \_ \text{ extends } \{\_\} \text{ is } \{\dots, m(x) = t, \dots\} \quad m \neq \text{init}}{\text{mbody}(m, C) = \lambda x.t} \text{ (MB-CLASS)}$	
	$\frac{\text{class } C : \_ \text{ extends } D\bar{j}\{\_\} \text{ is } \{\bar{M}\} \quad m \notin \bar{M}}{\text{mbody}(m, C) = \text{mbody}(m, D)} \text{ (MB-SUPER)}$	
$\text{q}(T) \text{ where q ::= un   lin}$	$\text{un}(\text{Top})$	$\frac{\text{not un}(T)}{\text{lin}(T)}$
$\text{classof}(T) = C$	$\text{class } C : \_ \text{ extends } \{\bar{f} : \_, \bar{m} : \Pi_. (\bar{T} \rightsquigarrow \bar{T} \times \_ \rightarrow \_) \text{ is } \{\_\}$	$\text{un}(T)$

Figure 7 Auxiliary functions and predicates.

The two rules S+L and S+R<sub>k</sub> together imply that a type  $T + U$  is a least upper bound of  $T$  and  $U$ . S- $\times$  expresses that the subtyping relation is a congruence. S-RECORD checks compatibility between field typings of the same class  $C$ .

### 3.3.2 Typing

**Auxiliary Functions and Predicates.** As in Featherweight Java (FJ) [25], our typing rules rely on a few auxiliary functions and predicates. These are given in Figure 7 and described below. We denote by  $\sqcup$  the disjoint union of field types, i.e. the operation of  $F_1 \sqcup F_2$  is defined by merging  $F_1$  and  $F_2$  if their domains are disjoint, being undefined otherwise. We write  $m \notin \bar{l}$  and  $m \notin \bar{M}$  to indicate that the method name  $m$  is not included, respectively, in the sequence of member names  $\bar{l}$  and method definitions  $\bar{M}$ . We denote by  $T[C\bar{i}/D]$  the substitution of  $C\bar{i}$  for the free occurrences of  $D$  bound to a type  $\rightsquigarrow$  in  $T$ .

The partial function  $\text{classof}(T)$  looks up the class of a type  $T$  of the form  $C\bar{i}$  and  $\Sigma a : I.U$ , being undefined for other forms. Both  $\text{fields}(T)$  and  $\text{mtype}(m, T)$ , also partial functions, look up member types. Notice that a subclass may extend an instantiated superclass, which

$\Delta_1; \Gamma \vdash r : T \dashv \Delta_2$	Under initial contexts $\Delta_1; \Gamma$ , path $r$ has type $T$ , with final context $\Delta_2$
$\frac{\Delta \vdash \Gamma}{\Delta; \Gamma, r : T \vdash r : T \dashv \Delta}$ (T-REF)	$\frac{\Delta; \Gamma \vdash r : C[F] \dashv \Delta}{\Delta; \Gamma \vdash r.f : F(f) \dashv \Delta}$ (T-FIELD)
$\frac{\Delta_1; \Gamma \vdash r : \Sigma a : I.T \dashv \Delta_2}{\Delta_1; \Gamma \vdash r : T \dashv \Delta_2, a : I}$ (T-UNPACK)	
$\frac{\Delta; \Gamma \vdash r : C[F] \dashv \Delta \quad \Delta \vdash C[F] <: \text{fields}(C.i)}{\Delta; \Gamma \vdash r : C.i \dashv \Delta}$ (T-HIDE)	

■ **Figure 8** Typing rules for paths.

means that, because of substitutions, the types of fields and methods in the subclass may not be identical to those in the superclass. On the other hand,  $\text{mbody}(m, C)$  is used only in the operational semantics. The predicate  $q(T)$  assigns a qualifier  $q$  to a type  $T$ : a type is said to be unrestricted (**un**) if it denotes an instance of a type invariant class, that is, a class whose methods do not change the state of the current object (the input and output types are the same); it is linear (**lin**) if its class defines at least one type varying method, which indicates that the state of the current object is modified.

**Term Typing.** For typing terms, we use a judgement of the form  $\Delta_1; \Gamma_1 * r_1 \vdash t : T \dashv \Delta_2; \Gamma_2 * r_2$  meaning that the evaluation of term  $t$  may both extend the context  $\Delta_1$  (for example, with existential variables that arise from the types of fields, or with propositions) and change the types contained in  $\Gamma_1$  (for example, by assigning values to objects, or by calling methods on them), giving rise to the final contexts  $\Delta_2; \Gamma_2$ . Linearity is yet another reason for a different final object context: if  $x$  is linear and is used in  $t$ , then  $x$  is consumed and does not appear in  $\Gamma_2$ . The judgement includes  $r_1$  and  $r_2$  in the style of Gay et al. [21], which are paths needed for typing runtime terms and tracing objects in the heap. When typechecking a program, both  $r_1$  and  $r_2$  are always `this`, and are used exclusively to access the fields of the current class. Hence, the judgement for typing top-level terms will always have the form  $\Delta_1; \Gamma_1, \text{this} : C[F_1] * \text{this} \vdash t : T \dashv \Delta_2; \Gamma_2, \text{this} : C[F_2] * \text{this}$ , where  $\Gamma_1$  and  $\Gamma_2$  differ only in the method parameter  $x$ . If  $\Gamma_1$  is  $x : U$  and  $U$  is linear, then  $\Gamma_2$  must be  $\epsilon$  since  $x$  has been consumed by  $t$ .

The typing rules for the top-level terms (Figure 4) are given in Figure 9. They use a judgement  $\Delta_1; \Gamma \vdash r : T \dashv \Delta_2$  for typing paths (Figure 8), and a definition  $C.l_k$  which means  $T_k$  for class  $C : \Delta$  extends  $T\{l_1 : T_1, \dots, l_n : T_n\}$  is  $\{\bar{M}\}$  with  $1 \leq k \leq n$ .

► **Definition 1** (Operations on Field Types and Object Contexts).

- If  $F = \{f_1 : T_1, \dots, f_n : T_n\}$ , then  $F(f_k) \triangleq T_k$ , and  $F\{f_j \leftrightarrow U\} \triangleq \{f_1 : T'_1, \dots, f_n : T'_n\}$  where  $T'_k = T_k$  and  $T'_j = U$  for  $k \neq j$  and  $n \geq 1$  and  $1 \leq k \leq n$  and  $1 \leq j \leq n$ .
- $(\Gamma, x : T)\{x \leftrightarrow U\} \triangleq \Gamma, x : U$ .
- $\Gamma\{r.f \leftrightarrow T\} \triangleq \Gamma\{r \leftrightarrow C[F\{f \leftrightarrow T\}]\}$  if  $\Delta; \Gamma \vdash r : C[F] \dashv \Delta$  for some  $\Delta$ .

We now comment on these rules: T-UNVAR and T-LINVAR are used to access a parameter. The former is the standard rule for reading a variable, while the latter implements destructive reads. T-UNFIELD is used for field access, being defined for unrestricted types only (since the effect of reading  $f$  linear would remove it from the current object type). T-NEW is the rule for object creation, giving the new object the type from the init method signature. T-ASSIGN

$\Delta_1; \Gamma_1 * r_1 \vdash t : T \dashv \Delta_2; \Gamma_2 * r_2$	Under initial contexts $\Delta_1; \Gamma_1$ with path $r_1$ , term $t$ has type $T$ , with final contexts $\Delta_2; \Gamma_2$ and path $r_2$
	$\frac{\Delta \vdash \Gamma \quad \mathsf{un}(T)}{\Delta; \Gamma, x : T * r \vdash x : T \dashv \Delta; \Gamma, x : T * r} (\text{T-UNVAR})$
	$\frac{\Delta \vdash \Gamma \quad \mathsf{lin}(T)}{\Delta; \Gamma, x : T * r \vdash x : T \dashv \Delta; \Gamma * r} (\text{T-LINVAR})$
	$\frac{\Delta; \Gamma \vdash r.f : T \dashv \Delta \quad \mathsf{un}(T)}{\Delta; \Gamma * r \vdash f : T \dashv \Delta; \Gamma * r} (\text{T-UNFIELD})$
	$\frac{\Delta \vdash \Gamma}{\Delta; \Gamma * r \vdash \mathbf{new} C() : C.\mathbf{init} \dashv \Delta; \Gamma * r} (\text{T-NEW})$
$\Delta_1; \Gamma_1 * r_1 \vdash t : T \dashv \Delta_2; \Gamma_2 * r_2$	
$\Delta_2; \Gamma_2 \vdash r_2 : C[F] \dashv \Delta_2 \quad \Delta_2; \Gamma_2 \{r_2.f \leftrightarrow T\} \vdash r_2 : C\bar{i} \dashv \Delta_2$	$\frac{\Delta_2; \Gamma_2 \vdash r_2 : C[F] \dashv \Delta_2 \quad \Delta_2; \Gamma_2 \{r_2.f \leftrightarrow T\} \vdash r_2 : C\bar{i} \dashv \Delta_2}{\Delta_1; \Gamma_1 * r_1 \vdash f := t : F(f) \dashv \Delta_2; \Gamma_2 \{r_2.f \leftrightarrow T\} * r_2} (\text{T-ASSIGN})$
$\Delta_1; \Gamma_1 * r_1 \vdash t_1 : U \dashv \Delta_2; \Gamma_2 * r_2 \quad \Delta_2; \Gamma_2 * r_2 \vdash t_2 : T \dashv \Delta_3; \Gamma_3 * r_2 \quad \mathsf{un}(U)$	$\frac{\Delta_1; \Gamma_1 * r_1 \vdash t_1 : U \dashv \Delta_2; \Gamma_2 * r_2 \quad \Delta_2; \Gamma_2 * r_2 \vdash t_2 : T \dashv \Delta_3; \Gamma_3 * r_2 \quad \mathsf{un}(U)}{\Delta_1; \Gamma_1 * r_1 \vdash t_1; t_2 : T \dashv \Delta_3; \Gamma_3 * r_2} (\text{T-SEQ})$
$\Delta_1; \Gamma_1 * r_1 \vdash t : U[\theta] \dashv \Delta_2; \Gamma_2 * r_2 \quad \Delta_2; \Gamma_2 \vdash r_2 : C\bar{i} \dashv \Delta_2$	
$\mathsf{mtype}(m, C\bar{i}) = \Pi \Delta. (C\bar{i} \rightsquigarrow T \times U \rightarrow W) \quad \Delta_2 \vdash \Delta : \theta$	
$\Delta_2; \Gamma_2 \{r_2 \leftrightarrow T[\theta]\} \vdash r_2 : C\bar{j} \dashv \Delta_3$	$\frac{\Delta_2; \Gamma_2 \{r_2 \leftrightarrow T[\theta]\} \vdash r_2 : C\bar{j} \dashv \Delta_3}{\Delta_1; \Gamma_1 * r_1 \vdash m(t) : W[\theta] \dashv \Delta_3; \Gamma_2 \{r_2 \leftrightarrow \mathsf{fields}(C\bar{j})\} * r_2} (\text{T-SELFCALL})$
$\Delta_1; \Gamma_1, r_1 : C[F] * r_1 \vdash t : U[\theta] \dashv \Delta_2; \Gamma_2 * r_2 \quad \Delta_2; \Gamma_2 \vdash r_2.f : T_1 \dashv \Delta_3$	
$\mathsf{mtype}(m, T_1) = \Pi \Delta. (T_1 \rightsquigarrow T_2 \times U \rightarrow W) \quad \Delta_3 \vdash \Delta : \theta$	
$\Delta_3; \Gamma_2 \{r_2.f \leftrightarrow T_2[\theta]\} \vdash r_2 : C\bar{i} \dashv \Delta_3$	$\frac{\Delta_3; \Gamma_2 \{r_2.f \leftrightarrow T_2[\theta]\} \vdash r_2 : C\bar{i} \dashv \Delta_3}{\Delta_1; \Gamma_1, r_1 : C[F] * r_1 \vdash f.m(t) : W[\theta] \dashv \Delta_3; \Gamma_2 \{r_2.f \leftrightarrow T_2[\theta]\} * r_2} (\text{T-CALL})$
$\Delta_1; \Gamma_1 \vdash r.f : (U_1 + U_2) \dashv \Delta_2 \quad \mathsf{classof}(U_k) = C_k$	
$\Delta_2; \Gamma_1 \{r.f \leftrightarrow U_k\} * r \vdash t_k : T \dashv \Delta_3; \Gamma_2 * r \quad C_1 \neq C_2$	$\frac{\Delta_2; \Gamma_1 \{r.f \leftrightarrow U_k\} * r \vdash t_k : T \dashv \Delta_3; \Gamma_2 * r \quad C_1 \neq C_2}{\Delta_1; \Gamma_1 * r \vdash \mathbf{case} f \mathbf{of} (C_k \Rightarrow t_k)_{k \in 1,2} : T \dashv \Delta_3; \Gamma_2 * r} (\text{T-CASE})$
$\Delta_1; \Gamma_1 * r_1 \vdash t : \mathbf{Boolean} p \dashv \Delta_2; \Gamma_2 * r_2$	
$\Delta_2, p; \Gamma_2 * r_2 \vdash t_1 : T \dashv \Delta_3; \Gamma_3 * r_2 \quad \Delta_2, \neg p; \Gamma_2 * r_2 \vdash t_2 : T \dashv \Delta_3; \Gamma_3 * r_2$	$\frac{\Delta_2, p; \Gamma_2 * r_2 \vdash t_1 : T \dashv \Delta_3; \Gamma_3 * r_2 \quad \Delta_2, \neg p; \Gamma_2 * r_2 \vdash t_2 : T \dashv \Delta_3; \Gamma_3 * r_2}{\Delta_1; \Gamma_1 * r_1 \vdash \mathbf{if} t \mathbf{then} t_1 \mathbf{else} t_2 : T \dashv \Delta_3; \Gamma_3 * r_2} (\text{T-IF})$
$\Delta_1; \Gamma_1 * r_1 \vdash t_1 : \mathbf{Boolean} p \dashv \Delta_2; \Gamma_2 * r_2$	
$\Delta_2, p; \Gamma_2 * r_2 \vdash t_2 : \mathbf{Top} \dashv \Delta_2; \Gamma_2 * r_2$	$\frac{\Delta_2, p; \Gamma_2 * r_2 \vdash t_2 : \mathbf{Top} \dashv \Delta_2; \Gamma_2 * r_2}{\Delta_1; \Gamma_1 * r_1 \vdash \mathbf{while} t_1 \mathbf{do} t_2 : \mathbf{Top} \dashv \Delta_2, \neg p; \Gamma_2 * r_2} (\text{T-WHILE})$
$\Delta_1; \Gamma_1 * r_1 \vdash t : U \dashv \Delta_2; \Gamma_2 * r_2 \quad \Delta_2 \vdash U <: T$	$\frac{\Delta_1; \Gamma_1 * r_1 \vdash t : U \dashv \Delta_2; \Gamma_2 * r_2 \quad \Delta_2 \vdash U <: T}{\Delta_1; \Gamma_1 * r_1 \vdash t : T \dashv \Delta_2; \Gamma_2 * r_2} (\text{T-SUB})$

■ **Figure 9** Typing rules for terms in the top-level language.

modifies a field of the current object, acting on its type  $C[F]$ . Unlike the rule for assignment in Java, when a field is changed, we need to check all the other fields in  $F$  to ensure that any dependencies are satisfied. We do this with judgement  $\Delta_2; \Gamma_2\{r_2.f \leftarrow T\} \vdash r_2 : C\bar{i} \dashv \Delta_2$  derived by rule T-HIDE that recovers a top-level type  $C\bar{i}$  using with the updated context as its initial context. Again, unlike the standard rule for assignment, our rule returns as its result the type of the old object contained in the field as part of the linear control of objects. T-SEQ is the standard rule for the sequence operation, except that it checks the first subterm and considers its possible effects in the typing context that checks the second one.

The two rules for calling methods are rather elaborate. T-SELF CALL checks the type of the parameter as usual, but uses rule T-HIDE to obtain a top-level type for the current object  $r$  of the form  $C\bar{i}$  that allows method  $m$  to be called (its signature yielding a substitution  $\theta$  applied to the parameter and output types). The final object context is updated in the conclusion with a type obtained by  $\text{fields}(C\bar{j})$  for  $r$ , where  $C\bar{j}$  is derived by possibly unpacking the receiver output type  $T[\theta]$  in the premise  $\Delta_2; \Gamma_2\{r_2 \leftarrow T[\theta]\} \vdash r_2 : C\bar{j} \dashv \Delta_3$ . T-CALL checks a method call on a field, combining the strategies used in T-ASSIGN and T-SELF CALL.

T-CASE makes the case distinction on a field  $f$  with a union type. Each branch is then typed with an initial context where  $f$  is bound to either the left or the right type. Two branches must have the same type and final contexts, because  $f$  can only be bound to one type. T-IF expects  $t$  in the condition to be of type Boolean  $p$ . Each branch is then typed with initial contexts asserting or negating the proposition  $p$ . T-WHILE is analogous to T-IF, yet simpler. T-SUB is the usual subsumption rule, adapted to our requirements.

**Program Typing.** A well-formed program relies on well-typed fields, methods and classes, which we formally define in Figure 10. The judgement  $\vdash_C M$  states that a method  $M$  in a class  $C$  is well-typed. T-METHOD constructs the judgement for checking the body of a regular method, whereas T-INIT initialises all fields, including the inherited ones. The judgement  $\Delta \vdash_T l : T$  checks that a member type is well-formed. T-FTYPE states that a field must be “typed” by kind  $\star$  of proper types. When checking method signatures, one of the following must hold: the method is altogether new (T-MTYPE), or a correct override of a superclass method (T-OVERRIDE). These judgements are used by T-CLASS. By T-PROGRAM a program is well-formed if each class defined in it is well-typed.

### 3.4 Operational Semantics

Figure 11 defines an operational semantics on states  $S$  the form  $(h * r, t)$  where the object path  $r$  is used to resolve field references appearing in the term  $t$ . As usual, we denote by  $t[o/x]$  the substitution of  $o$  for the free occurrences of  $x$  in  $t$  defined in the standard way.

► **Definition 2 (Operations on Heaps).** Let  $h$  be the heap of the form  $h = (h_0, o = R)$  where  $R$  is the object record  $C\{f_1 = o_1, \dots, f_n = o_n\}$ . Then,  $h(o) \triangleq R$  and  $h(o).\text{class} = C$  and for all  $k$  such that  $1 \leq k \leq n$ ,

- $R.f_k \triangleq o_k$ .
- $R\{f_j \leftarrow o\} \triangleq C\{\bar{f} = \bar{o}'\}$  where  $o'_k = o_k$  and  $o'_j = o$  for  $k \neq j$  and  $1 \leq j \leq n$ .
- $h\{o.f_k \leftarrow o'\} \triangleq (h_0, o = R\{f_k \leftarrow o'\})$ .
- $h(r) \triangleq o_k$  if  $r = o.f_k$ , and  $h\{r.f \leftarrow o'\} \triangleq h\{o_k.f \leftarrow o'\}$ .

R-NEW creates a fresh object and adds it to the heap, after having initialised all fields. For this, it relies on the reduction of states for sequenced object creation. R-ASSIGN replaces the value of a field  $f$  of the current object located at  $r$  with a new reference, and returns

$\vdash_C M$	Method $M$ is well-formed in class $C$
	class $C : \Delta_1$ extends $\_\{\dots, m : \Pi\Delta_2.(T_1 \rightsquigarrow T_2 \times U \rightarrow W), \dots\}$ is $\{\_\}$
	$\Delta_1, \Delta_2; x : U, \text{this} : \text{fields}(T_1) * \text{this} \vdash t : W \dashv \Delta_3; \Gamma, \text{this} : C[F] * \text{this}$
	$\frac{x : U \in \Gamma \Rightarrow \text{un}(U) \quad \Delta_3 \vdash C[F] <: \text{fields}(T_2) \quad m \neq \text{init}}{\vdash_C m(x) = t} \text{(T-METHOD)}$
	$\frac{\text{fields}(C.\text{init}) = C[F] \quad \epsilon; \epsilon * r \vdash \text{new } \bar{C}() : F(\bar{f}) \dashv \epsilon; \epsilon * r \quad \text{no cycles in } C}{\vdash_C \text{init}() = \bar{f} := \text{new } \bar{C}()} \text{(T-INIT)}$
$\Delta \vdash_T l : U$	Member $l$ has type $U$ with supertype $T$
	$\frac{\Delta \vdash U : \star \quad \text{mtype}(m, T) \text{ undefined}}{\Delta \vdash_T f : U} \text{(T-FTYPE)} \quad \frac{\Delta_1 \vdash \Pi\Delta_2.(\bar{C}i \times U \times W) : \star \quad \Delta_1, \Delta_2 \vdash Cj <: T_2}{\Delta_1 \vdash_T m : \Pi\Delta_2.(\bar{C}i \rightsquigarrow T_2 \times U \rightarrow W)} \text{(T-MTYPE)}$
	$\frac{\Delta_1 \vdash \Pi\Delta_2.(T_1 \times T_2 \times U' \times W) <: \Pi\Delta'_2.(T'_1 \times T'_2 \times U \times W')}{\Delta_1 \vdash_T m : \Pi\Delta_2.(T_1 \rightsquigarrow T_2 \times U \rightarrow W)} \text{(T-OVERRIDE)}$
$\vdash L$	Class declaration $L$ is well-formed
	$\frac{\Delta \vdash T : \star \quad \Delta \vdash_T \bar{l} : \bar{T} \quad \vdash_C \bar{M}}{\vdash \text{class } C : \Delta \text{ extends } T\{\bar{l} : \bar{T}\} \text{ is } \{\bar{M}\}} \text{(T-CLASS)}$
$\vdash P$	Program $P$ is well-formed
	$\frac{\vdash L_1 \quad \dots \quad \vdash L_n}{\vdash L_1 \dots L_n} \text{(T-PROGRAM)}$

■ **Figure 10** Typing rules for program formation.

the former object pointed by  $f$ . R-SEQ reduces to the second part of the sequence of terms, discarding the first part only after it has become an object.

R-SELF CALL is relative to a method call on the current object at  $r$ . The rule prepares the method body  $t$  with a substitution (the actual parameter for the formal one) before evaluating the term. R-CALL is the rule for a call on the object at  $f$  (relative to the current object at  $r$ ), being defined in a slightly different way. The rule makes  $r.f$  become the current object and wraps the method body  $t$ , prepared with the parameter substitution, in a return term that replaces the method call. Then, the body is reduced to an object in rule R-RETURN which also recovers the previous current object at  $r$ .

R-CASE $_k$  means that either branch is taken, with the first having precedence over the second, i.e. the second branch is only tried if the condition  $(h(r.f).\text{class} = C_1)$  fails. The two rules R-IFTRUE and R-IFFALSE use the special true and false objects for the references that control the condition. In rule R-WHILE, the term is rewritten to a nested conditional, using top for the body of the else branch. R-CONTEXT is standard for reduction in contexts, defining which term should be evaluated next.

$S_1 \longrightarrow S_2$  State  $S_1$  reduces to  $S_2$

$$\begin{array}{c}
 \text{mbody(init, } C) = \bar{f} := \text{new } \bar{C}() \\
 \frac{(h_1 * r, \text{new } \bar{C}()) \longrightarrow (h_2 * r, \bar{o}) \quad o \notin \text{dom}(h_2)}{(h_1 * r, \text{new } C()) \longrightarrow ((h_2, o = C\{\bar{f} = \bar{o}\} * r), o)} \text{ (R-NEW)} \\
 \frac{h(r).f = o_1}{(h * r, f := o_2) \longrightarrow (h\{r.f \leftarrow o_2\} * r, o_1)} \text{ (R-ASSIGN)} \\
 (h * r, o; t) \longrightarrow (h * r, t) \text{ (R-SEQ)} \quad \frac{h(r).\text{class} = C \quad \text{mbody}(m, C) = \lambda x.t}{(h * r, m(o)) \longrightarrow (h * r, t[o/x])} \text{ (R-SELFCALL)} \\
 \frac{h(r.f).\text{class} = C \quad \text{mbody}(m, C) = \lambda x.t}{(h * r, f.m(o)) \longrightarrow (h * r.f, \text{return } t[o/x])} \text{ (R-CALL)} \\
 (h * r.f, \text{return } o) \longrightarrow (h * r, o) \text{ (R-RETURN)} \\
 \frac{h(r.f).\text{class} = C_k}{(h * r, \text{case } f \text{ of } (C_k \Rightarrow t_k)_{k \in 1,2}) \longrightarrow (h * r, t_k)} \text{ (R-CASE}_k\text{)} \\
 (h * r, \text{if true then } t_1 \text{ else } t_2) \longrightarrow (h * r, t_1) \text{ (R-IFTRUE)} \\
 (h * r, \text{if false then } t_1 \text{ else } t_2) \longrightarrow (h * r, t_2) \text{ (R-IFFALSE)} \\
 \frac{t_2 = \text{if } t \text{ then } (t_1; \text{while } t \text{ do } t_1) \text{ else top}}{(h * r, \text{while } t \text{ do } t_1) \longrightarrow (h * r, t_2)} \text{ (R-WHILE)} \\
 \frac{(h_1 * r_1, t_1) \longrightarrow (h_2 * r_2, t_2)}{(h_1 * r_1, \mathcal{E}[t_1]) \longrightarrow (h_2 * r_2, \mathcal{E}[t_2])} \text{ (R-CONTEXT)}
 \end{array}$$

Figure 11 Reduction rules for states.

## 4 Type Soundness

In order to establish type soundness, we need an additional set of relations that describe heaps and runtime states. This is given in Figure 12. For typing the heap, we use a judgement of the form  $\Delta; \Gamma \vdash h$  that states that under contexts  $\Delta; \Gamma$  the heap  $h$  is well-formed. By rule T-EMPTYHEAP, a heap is constructed from typing contexts containing assumptions and types for all the objects relative to locations added to the heap by rule T-HEAP. The latter ensures that each heap entry has the prescribed field typing. The most important feature of this rule is that all aliases of linear references are explicitly forbidden by the rightmost premise. For typing sequenced objects  $\bar{o}$  as part of a runtime state, the judgement relies on T-UNVAR and T-LINVAR as appropriate to type each object. In particular, for each linear  $o_k$  in  $o_1, \dots, o_n$ , with  $1 \leq k \leq n$ , the initial typing context contains  $o_k$  and the final one of the extended heap does not, meaning that a heap that contains multiple references to the same linear object is not typable. The similar inverse argument justifies the existence of cyclic structures in the heap. Rule T-HEAPHIDE is used as needed in order to replace an internal object type by an equivalent top-level one (cf. [21]).

Finally, we use a judgement  $\Delta_1; \Gamma_1 \vdash S : T \dashv \Delta_2; \Gamma_2 * r$  to type states and formalize the main invariant of subject reduction. By T-STATE, given a state  $S$  of the form  $(h * r_1, t)$ , the heap  $h$  must be compatible with a context  $\Gamma_1$  under the assumptions in  $\Delta_1$ , which are the initial contexts that type the runtime term  $t$ , knowing from the leftmost premises that  $h$  is

$\Delta; \Gamma \vdash h$	Under contexts $\Delta; \Gamma$ , heap $h$ is well-formed
	$\frac{\Delta \vdash \Gamma}{\Delta; \Gamma \vdash \epsilon} (\text{T-EMPTYHEAP})$
$\Delta; \Gamma_1 \vdash h \quad \Delta; \Gamma_1 * o \vdash \bar{o} : F(\bar{f}) \dashv \Delta; \Gamma_2, o : C[F] * o$	$\Delta; \Gamma_2, o : C[F] \vdash h, (o = C\{\bar{f} = \bar{o}\}) \quad (\text{T-HEAP})$
$\Delta; \Gamma, o : C[F] \vdash h \quad \Delta; \Gamma, o : C[F] \vdash o : C\bar{i} \dashv \Delta$	$\Delta; \Gamma, o : C\bar{i} \vdash h \quad (\text{T-HEAPHIDE})$
$\Delta_1; \Gamma_1 \vdash S : T \dashv \Delta_2; \Gamma_2 * r$	Under initial contexts $\Delta_1; \Gamma_1$ , state $S$ has type $T$ , with final contexts $\Delta_2; \Gamma_2$ and path $r$
	$\frac{h \text{ complete} \quad \text{dom}(\Gamma_1) \subseteq \text{dom}(h) \quad \Delta_1; \Gamma_1 \vdash h \quad \Delta_1; \Gamma_1 * r_1 \vdash t : T \dashv \Delta_2; \Gamma_2 * r_2}{\Delta_1; \Gamma_1 \vdash (h * r_1, t) : T \dashv \Delta_2; \Gamma_2 * r_2} (\text{T-STATE})$

■ **Figure 12** Typing rules for heaps and states.

complete, i.e. for any  $o \in \text{dom}(h)$  we have  $\text{children}_h(o) \subseteq \text{dom}(h)$ , and that  $\text{dom}(\Gamma_1) \subseteq \text{dom}(h)$ , i.e. every object that has a type in  $\Gamma_1$  appears in  $h$  along with all of its children.

► **Definition 3** (Initial Heap and Object Context). In any well-formed program ( $\vdash P$ ),  $h_0$  and  $\Gamma_0$  represent the initial heap and object context such that  $h_0 = (\text{top} = \text{Top}\{\}, \text{false} = \text{Boolean}\{\}, \text{true} = \text{Boolean}\{\})$  and  $\Gamma_0 = (\text{top} : \text{Top}, \text{false} : \text{Boolean} \text{ false}, \text{true} : \text{Boolean} \text{ true})$ .

**Main Results.** By standard techniques [46], we now prove the expected results.

► **Theorem 4** (Subject Reduction). Suppose that  $P$  is a well-formed program ( $\vdash P$ ). In this context, let  $\Gamma_0 \subseteq \Gamma_1$  and  $h_0 \subseteq h_1$ , and  $S_1 = (h_1 * r_1, t)$ . If  $\Delta_1; \Gamma_1 \vdash S_1 : T \dashv \Delta_2; \Gamma_2 * r_2$  and  $S_1 \rightarrow S_2$ , then  $\Delta'_1; \Gamma'_1 \vdash S_2 : T \dashv \Delta'_2; \Gamma'_2 * r_2$  for some  $\Delta'_1, \Gamma'_1$  and  $\Gamma'_2$  such that  $\Delta_1 \subseteq \Delta'_1$  and  $\Gamma_2 \subseteq \Gamma'_2$ .

**Proof sketch.** In order to build this result, we need to prove a number of basic lemmas, namely inversion of the term typing relation, exchange for object contexts, weakening for index contexts, substitution for objects in term typing, substitution for indices, substitution for class types, and agreement of judgements. We also prove soundness and instantiation of function `mtype` as well as two lemmas (opening and closing) for the replacement of an internal object type by an equivalent top-level one when typing the heap. We then show that in well-formed DOL programs the types of objects describe their runtime values, that there never exists more than one reference to a linear object, and that all aliasing never produce a value of unexpected type. ◀

► **Theorem 5** (Progress). Suppose that  $P$  is a well-formed program ( $\vdash P$ ). In this context, let  $\Gamma_0 \subseteq \Gamma_1$  and  $h_0 \subseteq h_1$ .

1. If  $\Delta_1; \Gamma_1 \vdash (h_1 * r_1, t_1) : T \dashv \Delta_2; \Gamma_2 * r_2$ , then  $t_1$  is an object reference or  $(h_1 * r_1, t_1) \rightarrow (h_2 * r_2, t_2)$ .
2. If  $\Delta_1; \Gamma_1 \vdash (h_1 * r, \bar{t}) : \bar{T} \dashv \Delta_2; \Gamma_2 * r$ , then  $(h_1 * r, \bar{t}) \rightarrow (h_2 * r, \bar{t}')$ .

**Proof sketch.** By mutual induction on the structure of  $t$  and the length of  $\bar{t}$ . ◀

$\boxed{\Delta_1 \vdash T <: U \dashv \Delta_2}$  Under initial context  $\Delta_1$ , type  $T$  is a subtype of  $U$ , with final context  $\Delta_2$

$$\frac{\text{class } C : (\bar{a} : \bar{I}) \text{ extends } T\{\_\} \text{ is }\{\_\} \quad \Delta_1 \vdash T[\bar{i}/\bar{a}] <: D\bar{j} \dashv \Delta_2 \quad C \neq D}{\Delta_1 \vdash C\bar{i} <: D\bar{j} \dashv \Delta_2} \text{ (AS-SUPER)}$$

$$\frac{\Delta \triangleright T^\circ : \star}{\Delta \vdash T^\circ <: \text{Top} \dashv \Delta} \text{ (AS-TOP)}$$

$$\frac{\Delta_1 \vdash \bar{i} \equiv \bar{j} \dashv \Delta_2}{\Delta_1 \vdash C\bar{i} <: C\bar{j} \dashv \Delta_2} \text{ (AS-APP)}$$

$$\frac{\Delta_1, \hat{a} : I \vdash U[\hat{a}/a] <: T^\circ \dashv \Delta_2}{\Delta_1 \vdash \Pi a : I.U <: T^\circ \dashv \Delta_2} \text{ (AS-IIIL)} \quad \frac{\Delta_1, a : I \vdash T <: U \dashv \Delta_2}{\Delta_1 \vdash T <: \Pi a : I.U \dashv \Delta_2} \text{ (AS-IIIR)}$$

$$\frac{\Delta_1, a : I \vdash T <: U \dashv \Delta_2}{\Delta_1 \vdash \Sigma a : I.T <: U \dashv \Delta_2} \text{ (AS-SYL)} \quad \frac{\Delta_1, \hat{a} : I \vdash T^\circ <: U[\hat{a}/a] \dashv \Delta_2}{\Delta_1 \vdash T^\circ <: \Sigma a : I.U \dashv \Delta_2} \text{ (AS-SYR)}$$

$\boxed{\Delta_1; \Gamma_1 \vdash t \uparrow T \dashv \Delta_2; \Gamma_2}$  Under initial contexts  $\Delta_1; \Gamma_1$ , term  $t$  synthesizes type  $T$ , with final contexts  $\Delta_2; \Gamma_2$

$\boxed{\Delta_1; \Gamma_1 \vdash t \downarrow T \dashv \Delta_2; \Gamma_2}$  Under initial contexts  $\Delta_1; \Gamma_1$ , term  $t$  checks against input type  $T$ , with final contexts  $\Delta_2; \Gamma_2$

$$\Delta_1; \Gamma_1 \vdash \text{this}.f \uparrow T_1 \dashv \Delta_2 \quad \text{mtype}(m, T_1) = \Pi(\bar{a} : \bar{I}).(T_1 \rightsquigarrow T_2 \times U \rightarrow W)$$

$$\Delta_2, \bar{a} : \bar{I}; \Gamma_1 \vdash t \downarrow U[\bar{a}/\bar{a}] \dashv \Delta_3; \Gamma_2 \quad \bar{a} \text{ fresh}$$

$$\frac{\Delta_3; \Gamma_2 \{ \text{this}.f \leftrightarrow T_2[\bar{a}/\bar{a}] \} \vdash \text{this} \uparrow_h C\bar{i} \dashv \Delta_4}{\Delta_1; \Gamma_1 \vdash f.m(t) \uparrow W[\bar{a}/\bar{a}] \dashv \Delta_3; \Gamma_2 \{ \text{this}.f \leftrightarrow T_2[\bar{a}/\bar{a}] \}} \text{ (AT-CALL)}$$

$$\Delta_1; \Gamma_1 \vdash \text{this}.f \uparrow (U_1 + U_2) \dashv \Delta_2 \quad \text{classof}(U_k) = C_k$$

$$\frac{\Delta_2; \Gamma_1 \{ \text{this}.f \leftrightarrow U_k \} \vdash t_k \uparrow T_k \dashv \Delta_{k+2}; \Gamma_{k+2} \quad C_1 \neq C_2}{\Delta_1; \Gamma_1 \vdash \text{case } f \text{ of } (C_k \Rightarrow t_k)_{k \in 1,2} \uparrow (T_1 + T_2) \dashv (\Delta_3; \Gamma_3 \cdot \Delta_4; \Gamma_4)} \text{ (AT-CASE)}$$

**Figure 13** Selected algorithmic rules. In the subtyping rules,  $T^\circ$  means that  $T$  is not a type  $\Pi, \Sigma$ , or  $+$ . In rules AS-IIIL and AS-SYR, index variable  $\hat{a}$  is fresh.

## 5 Algorithmic Typechecking

We develop an algorithmic system in two steps. The first step is to introduce an existential index variable (written  $\hat{a}$  with the hat in the style of Dunfield and Krishnaswami [14, 15, 16]) into the initial index context whenever there is a need to make a guess at the appropriate index term  $i$ . The oracular rules in the declarative system are S-IIIL and S-SYR, T-HIDE, T-SELFCALL, T-CALL and T-MTYPE. In the algorithmic type system, each declarative judgement has a corresponding algorithmic judgement that takes an initial index context and yields a final index context, possibly augmented with knowledge about what index terms have to be. Instead of guessing, the algorithmic system adds judgements to instantiate existential index variables of the form  $\Delta_1 \vdash \hat{a} := i \dashv \Delta_2$ , and to equate index terms, namely  $\Delta_1 \vdash i \equiv j \dashv \Delta_2$ . The second step is to apply bidirectional typechecking [39] in order to distinguish rules that synthesize types from those that check terms against types already known, a technique that easily supports subtyping and index refinements. In the process, we also eliminate the nondeterminism associated with the subtyping and typing rules for paths.

The system uses the syntax and meta-variables of the declarative system (Figures 4 and 5). In addition to *solved* existential index variable declarations  $a : I$ , index contexts in the algorithmic system may also contain *unsolved* existential index variable declarations  $\hat{a} : I$ . Similarly to index contexts in the declarative system, index contexts in the algorithmic system are ordered sequences.

We give some of the algorithmic rules in Figure 13. The algorithmic subtyping rules use judgements of the form  $\Delta_1 \vdash T <: U \dashv \Delta_2$  where the final index context  $\Delta_2$  may carry information about solved existential index variables. In the bidirectional typechecking algorithm [39], we alternate between synthesizing types and checking terms against types already known. Bidirectional typechecking in DOL is formalised by replacing the typing judgement of the form  $\Delta_1; \Gamma_1 * r_1 \vdash t : T \dashv \Delta_2; \Gamma_2 * r_2$  with the following two judgements:  $\Delta_1; \Gamma_1 \vdash t \uparrow T \dashv \Delta_2; \Gamma_2$  for synthesizing and  $\Delta_1; \Gamma_1 \vdash t \downarrow T \dashv \Delta_2; \Gamma_2$  for checking.

► **Definition 6** (Complete Index Contexts). A complete index context, denoted by  $\phi$ , is an algorithmic index context such that for every existential index variable  $\hat{a}$  in  $\text{dom}(\phi)$ ,  $\phi(\hat{a}) \triangleq \hat{a} : I \doteq i$ .

**Soundness.** To show that the algorithmic system is sound with respect to the original system, we are given an algorithmic judgement, with an initial index context  $\Delta_1$  and a final index context  $\Delta_2$ , and  $\phi$  as a solved extension of context  $\Delta_2$ , and hence  $\text{dom}(\Delta_1) \subseteq \text{dom}(\phi)$  (by transitivity). Applying  $\phi$  as a substitution to the given algorithmic judgement produces a declarative judgement, which is the result we want to obtain.

► **Theorem 7** (Soundness of Algorithmic Subtyping). *If  $\Delta_1 \triangleright T : \star$  and  $\Delta_1 \triangleright U : \star$  and  $T[\Delta_1] = T$  and  $U[\Delta_1] = U$  and  $\Delta_1 \vdash T <: U \dashv \Delta_2$  and  $\phi$  extends  $\Delta_2$ , then  $\Delta_2[\phi] \vdash T[\phi] <: U[\phi]$ .*

► **Theorem 8** (Soundness of Algorithmic Typing). *Let  $\phi$  be a complete index context that extends  $\Delta_2$ .*

1. *If  $\Delta_1; \Gamma \vdash r \uparrow T \dashv \Delta_2$ , then  $\Delta_1[\phi]; \Gamma[\phi] \vdash r : T[\phi] \dashv \Delta_2[\phi]$ .*
2. *If  $\Delta_1; \Gamma_1 \vdash t \uparrow T \dashv \Delta_2; \Gamma_2$ , then  $\Delta_1[\phi]; \Gamma_1[\phi] * \text{this} \vdash t : T[\phi] \dashv \Delta_2[\phi]; \Gamma_2[\phi] * \text{this}$ .*
3. *If  $\Delta_1; \Gamma_1 \vdash t \downarrow T \dashv \Delta_2; \Gamma_2$  and  $\Delta_1 \triangleright T : \star$ , then  $\Delta_1[\phi]; \Gamma_1[\phi] * \text{this} \vdash t : T[\phi] \dashv \Delta_2[\phi]; \Gamma_2[\phi] * \text{this}$ .*

**Completeness.** To prove completeness of the algorithmic system, we somehow do the reverse of soundness: from a declarative derivation, which has no existential index variables, we obtain a complete index context along an algorithmic derivation. In completeness of algorithmic *subtyping*, we are given an initial index context  $\Delta_1$  and a complete index context  $\phi_1$  that extends it. In completeness of algorithmic *typing*, in addition we are given a final context  $\Delta'_1$  that may extend  $\Delta_1$  (with the result of unpacking or with propositions, for example) such that  $\text{dom}(\Delta_1) \subseteq \text{dom}(\Delta'_1)$  and  $\text{dom}(\Delta'_1) = \text{dom}(\phi_1)$ , and hence  $\text{dom}(\Delta_1) \subseteq \text{dom}(\phi_1)$ . We show that we can build an algorithmic derivation with a final context  $\Delta_2$ . However, the algorithmic rules generate fresh index variables that may not be in  $\Delta_1, \Delta'_1$  or  $\phi_1$ . So, completeness will also produce a complete index context  $\phi_2$  that extends both  $\Delta_2$  and  $\phi_1$  such that  $\text{dom}(\Delta_2) = \text{dom}(\phi_2)$ .

► **Theorem 9** (Completeness of Algorithmic Subtyping). *Let  $\phi_1$  be a complete index context that extends  $\Delta_1$  such that  $\text{dom}(\Delta_1) = \text{dom}(\phi_1)$ . If  $\Delta_1[\phi_1] \vdash T[\phi_1] : \star$  and  $\Delta_1[\phi_1] \vdash U[\phi_1] : \star$  and  $\Delta_1[\phi_1] \vdash T[\phi_1] <: U[\phi_1]$ , then  $\Delta_1 \vdash T[\Delta_1] <: U[\Delta_1] \dashv \Delta_2$  and there exists  $\phi_2$  that extends both  $\Delta_2$  and  $\phi_1$  such that  $\text{dom}(\Delta_2) = \text{dom}(\phi_2)$ .*

► **Theorem 10** (Completeness of Algorithmic Typing). *Let  $\phi_1$  and  $\Delta'_1$  be index contexts that extend  $\Delta_1$  such that  $\text{dom}(\Delta_1) \subseteq \text{dom}(\Delta'_1)$  and  $\text{dom}(\Delta'_1) = \text{dom}(\phi_1)$ .*

1. *If  $\Delta_1[\phi_1] \vdash T[\phi_1] : \star$ ,  $\Delta_1[\phi_1]; \Gamma[\phi_1] \vdash r : T[\phi_1] \dashv \Delta'_1[\phi_1]$ , then  $\Delta_1; \Gamma[\Delta_1] \vdash r : T[\Delta_1] \dashv \Delta_2$  and there exists  $\phi_2$  that extends both  $\Delta_2$  and  $\phi_1$  such that  $\text{dom}(\Delta_2) = \text{dom}(\phi_2)$ .*

2. If  $\Delta_1[\phi_1] \vdash T[\phi_1] : \star$  and  $\Delta_1[\phi_1]; \Gamma_1[\phi_1]*r_1 \vdash t : T[\phi_1] \dashv \Delta'_1[\phi_1]; \Gamma_2[\phi_1]*r_2$ , then depending on  $t$  either  $\Delta_1; \Gamma_1[\Delta_1] \vdash t \uparrow T[\Delta_1] \dashv \Delta_2$ ;  $\Gamma_2[\Delta_1]$  or  $\Delta_1; \Gamma_1[\Delta_1] \vdash t \downarrow T[\Delta_1] \dashv \Delta_2$ ;  $\Gamma_2[\Delta_1]$  and there exists  $\phi_2$  that extends both  $\Delta_2$  and  $\phi_1$  such that  $\text{dom}(\Delta_2) = \text{dom}(\phi_2)$ .

**Implementation.** Our prototype ships with an IDE developed as an Eclipse plugin based on the Xtext framework, where the examples can be typechecked, compiled and run. The IDE support includes: a code editor assistant for DOL programs, on-the-fly error checking, and target code generation in the form of Java classes. Our typechecker is a direct implementation of the algorithmic type system, extended with integer and boolean literals, local variables and all the syntactic sugar from the examples. Constraint checking is performed as part of typechecking via a direct interface to the Z3 constraint solver [12].

## 6 Related Work and Discussion

At the basis of index refinements lies the notion of dependent type developed by Martin-Löf [32], and first applied to proof assistants (logical frameworks) such as AUTOMATH [44], the Calculus of Constructions [11], NuPRL [10], Lego [54] and the Edinburgh Logical Framework [22]. While full dependent types are an appealing feature to integrate in programming languages, the price is increased complexity of typechecking. Unlike index refinements, full dependent types do not restrict the domain of variables appearing in types. When added to (possibly nonterminating) programming languages, the task of determining type equivalence becomes as difficult as determining term equivalence (which is undecidable in general).

Some programming languages offer different strategies to handle nonterminating programs. Cayenne [2] is a functional programming language in the style of Haskell with an undecidable dependent type system. A semi-decidable approach forces the typechecker to terminate within a number of prescribed steps, eventually providing the user with an answer. Epigram [33] builds on a tactic-driven proof engine, similar to that of the Coq proof assistant, requiring correctness proofs to be specified. Unlike Cayenne, Epigram rules out general recursive programs, avoiding nontermination and any form of effects, thus making typechecking decidable. Recursion is supported by the structure of dependent types which are inductive families with inductive indices.

The Ynot tool is an extension of the functional dependently-typed language included in Coq with support for side-effects via Hoare Type Theory (HTT) and Separation Logic [35, 36]. HTT introduces an indexed monadic type in the style of a Hoare triple to reason about mutation. While DOL’s varying types may have similarities with the Hoare type, our approach does not involve the complexity of higher-order abstraction. As HTT, the F\* language [43], designed for program verification, employs the monad technique generalising it to multiple monads. This ML-style functional language uses dependent and refinement types to specify effectful programs, and supports automated and interactive proofs. A related approach is provided by RSP1 [45] that allows programming with proofs in an imperative setting. The language offers decidable typechecking by banning impure operations from types with the purpose of letting the user prove arbitrary properties of programs. All these languages provide SMT-based automation and handle effectful programming. In that regard, they are close to DOL, yet they differ substantially in their aim to combine programming and theorem proving, which our language does not support. Targeting the C programming language, Deputy [9] also handles mutation using a Hoare-inspired typing rule ensuring that assignment results in a well-typed state. For decidability, Deputy combines compile time and runtime checking, as opposed to our approach in which typechecking is performed statically.

Index refinements as formulated by Xi and Pfenning [53] reduce typechecking to a constraint satisfaction problem on terms belonging to index sorts. Their approach (which we adopt) 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 [48], a language with a C-like syntax combining imperative programming with index refinements, and ATS [50] which also supports DML-style dependent types and linear types (named viewpoints). While closely related, DOL extends the ideas of Xanadu to class-based objects that exhibit state and behaviour. Our language also handles object-oriented programming features such as modular development, inheritance with subtyping, which Xanadu does not deal with. A proposal for building an object-oriented system on top of DML was formulated by Xi [49]. The language includes inheritance without subtyping, simulated via existentially quantified dependent types. Xi's object model is simpler than ours, since objects are not regarded as records of fields (they merely respond to messages), and the language does not include imperative features.  $\Omega$ mega [41] and Liquid Types [40] offer two more examples of functional languages with a strict phase separation; the latter is implemented in DSolve, a tool that automatically infers dependent types from an OCaml program and a set of logical qualifiers. Cyclone [26] is a type-safe extension of the C programming language, combining static analysis and runtime checks. It offers domain-specific indexed types for the purpose of safe multi-threading and memory management.

Another reference is Dependent JavaScript (DJS) [7], which introduces refinement types with predicates from an SMT-decidable logic in a dynamic real-world language. In DJS, imperative updates involve the presence of mutation: the types of variables are changed by assignment, for instance. The challenge is handled using *flow-sensitive heap types*, which allow tracking variable types, in combination with refinement types. The result is an increase in the language expressiveness by using type annotations inside JavaScript comments that account for side-effects. DJS employs the *alias types* approach [42] for strong updates in combination with thawing/freezing locations, an alternative to DOL's linear approach.

Other forms of dependent types include X10's constrained types [38], designed around the notion of constraints on the *immutable* state of objects. The core language proposed extends the purely functional FJ [25]. While appealing, constrained types currently cannot enforce invariants on the mutable state of objects. Dependent classes [20] provide another approach in the object-oriented setting. A class can be seen as forming a family of collaborating objects, much like a type family in traditional dependent type theory. The model is complex, since it also involves inheritance, and type soundness is hard to prove. Like DOL, full dependent classes and its lightweight version [27] support class-based programming and inheritance. A similar model is provided by Scala's path-dependent types [1] that unify nominal and structural type systems by allowing objects to contain type members. Dependent types in this model are expressed not in type signatures but in type placements. An abstract type refers to a type that must be defined by subclasses, becoming dependent on the instance it refers to. None of these languages supports an imperative style of programming, whereas DOL is designed to handle both mutable and immutable objects.

A kind of *typestate* seems to arise from having state exposed in (method) types and relying on input and output types that define pre- and post-conditions. In this sense, DOL relates to recent work on a typestate-oriented programming language [19] by Garcia et al. As DOL, Featherweight Typestate (FT) is a nominal object-oriented language with mutable state but whose types are enriched with state permissions. However, there are important technical differences between FT and DOL. The former is based on transitions that specify sequences of method calls explicitly, whereas DOL's method availability is less explicit. FT also allows flexible aliasing control by way of *access permissions* specified in types, whereas DOL uses only linear objects (adding a better alias control is seen as an orthogonal issue).

**Extensions.** We have left out of DOL’s formalisation some features that are desirable in practice. In particular, we need (1) richer index languages in domains of interest, possibly at the cost of decidable typechecking, and (2) alternatives to the current strategy for handling aliases. To relax the notion of uniqueness, we could, for instance, introduce an indirection from the main type context to a compile-time heap of objects in the style of alias types [42]. The possibility of aliasing indexed types would require a pointer to the object, allowed to be freely duplicated, while the type describing the object state must remain linear. However, to record type indirections, this approach would require additional type annotations.

---

### References

---

- 1 Nada Amin, Tiark Rompf, and Martin Odersky. Foundations of path-dependent types. In *OOPSLA*. ACM Press, 2014.
- 2 Lennart Augustsson. Cayenne a language with dependent types. In *ICFP*, pages 239–250. ACM Press, 1998.
- 3 Hendrik Pieter Barendregt. *The Lambda Calculus – Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.
- 4 Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of Agda – A functional language with dependent types. In *TPHOLs*, volume 5674 of *LNCS*, pages 73–78. Springer, 2009.
- 5 Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, 2013.
- 6 Patrice Chalin and Perry R. James. Non-null references by default in Java: Alleviating the nullity annotation burden. In *ECOOP*, pages 227–247. Springer, 2007.
- 7 Ravi Chugh, David Herman, and Ranjit Jhala. Dependent types for JavaScript. In *OOPSLA*, pages 587–606. ACM Press, 2012.
- 8 Maciej Cielecki, Jędrzej Fulara, Krzysztof Jakubczyk, and Lukasz Jancewicz. Propagation of jml non-null annotations in Java programs. In *Principles and Practice of Programming in Java*, pages 135–140. ACM Press, 2006.
- 9 Jeremy Condit, Matthew Harren, Zachary R. Anderson, David Gay, and George C. Necula. Dependent types for low-level programming. In *ESOP*, volume 4421 of *LNCS*, pages 520–535. Springer, 2007.
- 10 Robert L. Constable, Stuart F. Allen, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, Scott F. Smith, James T. Sasaki, and S. F. Smith. *Implementing mathematics with the Nuprl proof development system*. Prentice Hall, 1986.
- 11 Thierry Coquand and Gérard P. Huet. The calculus of constructions. *Inf. Comput.*, 76(2/3):95–120, 1988.
- 12 Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
- 13 Krishna Kishore Dhara and Gary T. Leavens. Forcing behavioral subtyping through specification inheritance. In *International Conference on Software Engineering*, pages 258–267, 1996.
- 14 Joshua Dunfield. *A Unified System of Type Refinements*. PhD thesis, Carnegie Mellon University, 2007. CMU-CS-07-129.
- 15 Joshua Dunfield and Neelakantan R. Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *ICFP*, pages 429–442. ACM Press, 2013.
- 16 Joshua Dunfield and Neelakantan R. Krishnaswami. Sound and complete bidirectional typechecking for higher-rank polymorphism with existentials and indexed types. *CoRR*, abs/1601.05106, 2016.

- 17 Manuel Fähndrich and K. Rustan M. Leino. Declaring and checking non-null types in an object-oriented language. In *OOPSLA*, pages 302–312. ACM Press, 2003.
- 18 Cormac Flanagan. Hybrid type checking. In *POPL*, pages 245–256, 2006.
- 19 Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. Foundations of typestate-oriented programming. *ACM Trans. Program. Lang. Syst.*, 36(4):12:1–12:44, 2014.
- 20 Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. Dependent classes. In *OOPSLA*, pages 133–152. ACM Press, 2007.
- 21 Simon J. Gay, Nils Gesbert, António Ravara, and Vasco Thudichum Vasconcelos. Modular session types for objects. *Logical Methods in Computer Science*, 11(4), 2015.
- 22 Robert Harper, Furio Honsell, and Gordon D. Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, 1993.
- 23 Tony Hoare. Null references: The billion dollar mistake. QCon, 2009.
- 24 Atsushi Igarashi and Hideshi Nagira. Union types for object-oriented programming. In *Symposium on Applied Computing*, pages 1435–1441. ACM Press, 2006.
- 25 Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *TOPLAS*, 23(3):396–450, 2001.
- 26 Trevor Jim, J. Gregory Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *USENIX*, pages 275–288. USENIX, 2002.
- 27 Tetsuo Kamina and Tetsuo Tamai. Lightweight dependent classes. In *GPCE*, pages 113–124. ACM Press, 2008.
- 28 Kenneth L. Knowles. *Executable Refinement Types*. PhD thesis, University of California, 2014.
- 29 Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006.
- 30 K. Rustan M. Leino. Extended static checking: A ten-year perspective. In *Informatics – 10 Years Back. 10 Years Ahead*, volume 2000 of *LNCS*, pages 157–175. Springer, 2001.
- 31 Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *TOPLAS*, 16(6):1811–1841, 1994.
- 32 Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis-Napoli, 1984.
- 33 Conor McBride. Epigram: Practical programming with dependent types. In *Advanced Functional Programming*, volume 3622 of *LNCS*, pages 130–170. Springer, 2004.
- 34 Conor McBride. How to keep your neighbours in order. In *ICFP*, pages 297–309. ACM Press, 2014.
- 35 Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. Ynot: dependent types for imperative programs. In *ICFP*, pages 229–240, 2008.
- 36 Aleksandar Nanevski, J. Gregory Morrisett, and Lars Birkedal. Hoare type theory, polymorphism and separation. *Journal of Functional Programming*, 18(5-6):865–911, 2008.
- 37 Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, 2007.
- 38 Nathaniel Nystrom, Vijay Saraswat, Jens Palsberg, and Christian Grothoff. Constrained types for object-oriented languages. In *OOPSLA*, pages 457–474. ACM Press, 2008.
- 39 Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, 2000.
- 40 Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In *PLDI*, pages 159–169. ACM Press, 2008.
- 41 Tim Sheard and Nathan Linger. Programming in Omega. In *Central European Functional Programming School*, volume 5161 of *LNCS*, pages 158–227. Springer, 2007.

- 42 Frederick Smith, David Walker, and J. Gregory Morrisett. Alias types. In *ESOP*, volume 1782 of *LNCS*, pages 366–381. Springer, 2000.
- 43 Nikhil Swamy, Cătălin Hrițcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F\*. In *POPL*, pages 256–270. ACM Press, 2016.
- 44 Diederik T. van Daalen. *The Language Theory of Automath*. PhD thesis, Technische Hogeschool Eindhoven, Eindhoven, 1980.
- 45 Edwin Westbrook, Aaron Stump, and Ian Wehrman. A language-based approach to functionally correct imperative programming. In *ICFP*, LNCS, pages 268–279. ACM Press, 2005.
- 46 Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.
- 47 Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, Pittsburgh, 1998.
- 48 Hongwei Xi. Imperative programming with dependent types. In *LICS*, pages 375–387. IEEE Press, 2000.
- 49 Hongwei Xi. Unifying object-oriented programming with typed functional programming. In *PEPM*, pages 117–125. ACM Press, 2002.
- 50 Hongwei Xi. Applied type system: Extended abstract. In *TYPES*, pages 394–408. Springer, 2004.
- 51 Hongwei Xi. Dependent ML: an approach to practical programming with dependent types. *Journal of Functional Programming*, 17(2):215–286, 2007.
- 52 Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *PLDI*, pages 249–257. ACM Press, 1998.
- 53 Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *POPL*, pages 214–227. ACM Press, 1999.
- 54 Luo Zhaojun and Robert Pollack. The LEGO proof development system: A user’s manual. Technical Report ECS-LFCS-92-211, University of Edinburgh, 1992.