Imperative Objects with Dependent Types

Joana Campos and Vasco T. Vasconcelos

Lasige and Department of Informatics
University of Lisbon, Portugal

Abstract. This paper proposes a dependent type theory for imperative objects. Building on state-of-the-art techniques for functional languages, we extend a class-based object-oriented language to support type dependencies which can express and statically enforce fine grained object invariants dependent on runtime data. Our classes, parameterised over index terms, provide a natural and elegant way to reason about and describe the consistency of object-oriented data structures. Methods assume all class invariants and may depend on further index terms to more accurately express conditions that the inputs (including the distinguished variable this) should satisfy, as well as expected side effects on this and other objects passed as arguments to the method. Intuitively, invariants and pre- and postconditions are included in the language via dependent types, while type checking remains decidable. We illustrate the type theory showing how classes can be modelled using our dependent type system. Our main results include soundness via subject reduction and progress properties.

Keywords: Dependent types, object-oriented, classes, mutable objects

1 Introduction

In object oriented languages, data structures are assumed to satisfy certain semantic properties. Classes representing object types use the notion of invariant to describe that an object is in a consistent state, at least at the beginning and end of any method execution. One of the particularly challenging aspects of invariant description is that many properties are not fixed at design time, but are dependent on runtime data.

Consider in Figure 1 a view of a bank account interface specified in Java, and below the same interface defined in Dependent Object-Oriented Language (DOL), the language proposed in this paper. There are several questions left unanswered by the Java interface. For example, what is the effect of the deposit and withdraw methods on the current bank account balance? A natural number is supposed to be passed to these methods, but what happens if a negative integer is passed instead? What happens when the the argument passed to withdraw is less than the current balance? (Does the method silently abort the call? Does it throw an exception?)
While in Java informal comments clear up some of these questions, in DOL the bank account type is precise enough to fully describe its behaviour. DOL uses a twofold strategy that consists of (1) tracking state with special index variables, declared at the class level, and (2) tracking state modifications explicitly with pre- and post-types (the relation is denoted by the wavy arrow \( \Rightarrow \), which may be omitted when types do not change). In the example, Account 0 is the type of a new instance, created with an initial balance set to 0. In DOL, a type Account \( b \) is said to be well-formed if the typechecker can prove that \( b \) is a term of type \texttt{integer}. The signatures of the deposit and withdraw methods propagate information about constraints on the state and the effect of the call on the method’s arguments, including the distinguished variable \texttt{this}. Note that each method is defined with an explicit first argument representing the caller of the method (implicitly provided by the call). Hence, the deposit method (lines 4–5) expects to be called on an object of type Account \( b \), where \( b:\texttt{integer} \), ensuring that its resulting type is Account \( (b+m) \), where \( m \) is some natural number. Index type \texttt{natural} is short for a refinement type of the form \( \{x:\texttt{integer} \mid 0 \leq x\} \). The withdraw method (lines 6–7) introduces an index refinement over natural numbers, constraining the amount to be withdrawn to be less than or equal to the current balance. This simply means that in DOL it is a type error to call the withdraw method without satisfying this condition.

As this simple example suggests, dependent types may be useful in general purpose programming, and in particular within the object-oriented paradigm. The idea of DOL is that object-oriented programmers may use dependent types as much as they find useful to capture correctness properties and express some aspects of a program’s behaviour. The burden of a dependent type system is paid off with the benefits of precise abstractions and formal documentation that is statically verified by the compiler.
For the example above to be practical, we need to ensure that type compatibility remains decidable. For example, an object of type `Account (500+120−20)` is expected to be also of type `Account 600`. Since deciding the equivalence of two arbitrary program terms is, in general, undecidable, the challenge with dependent types is that typechecking algorithms must find a way to tackle this problem. Two fundamental approaches have been proposed. Some languages and proof assistants, such as Cayenne [2], Epigram [15], Coq [6] or Agda [17], provide full dependent types, which means that all the programs are allowed to appear in types. While very expressive, these languages tend, in general, to be too expensive for use in realistic programs, namely by disallowing any form of effects, employing some kind of analysis that force programs to terminate, or requiring a great deal of manual proof annotations to guide the typechecker.

As an alternative to full dependency, restricted, less expressive forms of dependent types have been proposed in functional languages such as Dependent ML (DML) [26,31], ATS [28], Ωmega [22], or DSolve (Liquid Types) [21]. The crucial idea is to introduce a clear separation between compilation and computation through type dependencies on special terms, often called index terms, that are used exclusively to reason about values in the program. The question of typechecking decidability is thus reduced to a constraint satisfaction problem, which may be solved by automatic constraint solvers associated with the domain in use. By adopting this strategy, only equality of indices, not of arbitrary terms, needs to be proved. To ensure termination, solutions include the restriction of indices to finite domains (often relations over integer numbers), and of predicates to linear inequalities [31].

DOL adopts the second approach, since the phase distinction is central to support decidable typechecking. The language combines practical dependent types from DML with a subset of the expected features from object-oriented programming, such as classes, inheritance, objects and methods that operate on objects to change state. The concepts of object invariant, used to describe the consistent states of an object, and of pre- and postconditions for methods, complementing the object-level specification, are supported using indexed types.

As illustrated in the example above, DOL classes and methods can be seen as introducing universal dependent types. However, existential dependent types are essential to express many examples in practical programming with indefinite result types [31]. For example, a `filter` method on a list requires an existential type to express the fact that the size of the resulting list is unknown. Therefore, DOL also includes existential types in methods.

In order to be able to describe in types the state of objects, our language adopts the “pure” object-oriented programming model, similar to the Smalltalk or Ruby object models. In DOL, even an integer number is an instance of the `Integer` class, and behaves exactly like any other object, responding to methods available in the class. This motivates three fundamental design decisions: (1) object references are the only values in our language (as opposed to, say, Java that includes primitive values), hence (2) methods use a call-by-reference strategy only, and (3) state-modifying methods which do not return values (similar to
void methods) are the only kind of methods that can be defined in DOL. In the examples, we define some syntactic sugar to make these decisions less obvious to the programmer.

The central contributions of this work are the following:

– The study of universal and existential dependent types for a class-based, imperative object-oriented language. The integration of dependent types permits properties to be specified at both the class and method levels for constraining object types and reasoning statically about values in a program.
– The introduction of a relationship between the input and output types of a method parameter, similar to pre- and post-conditions specifications, for the purpose of tracking state modifications. This allows us to reason about mutable state. Combined with dependent types, it is the contribution that mostly distinguishes our approach from other work on dependent types.
– The treatment of single inheritance in classes with dependent types. A class may extend another class, as long as it satisfies the constraints of the superclass. This involves defining the standard subtyping relation for refinement types [13]. Support for subclasses also means that programmers can define inductive structures in DOL.
– The definition of a practical type system for objects. By adopting a phase separation, similar to the one proposed in DML, we use type annotations as an approximation to the operational behaviour of programs and rely on decidable domains to automatically solve constraints. Hence, we simplify the equivalence relation on terms, and the amount of annotations required to guide the typechecker.

The rest of the paper is organised as follows. Section 2 discusses related work, introducing some of the main ideas which are illustrated in Section 3 through some motivating examples. Section 4 formalises the core language. The operational semantics is described in Section 5, and Section 6 presents the type system. In Section 7, we prove the key properties of the type system, the main result being type soundness via progress and subject reduction. Finally, we conclude in Section 8, outlining future work. The complete formal definition of DOL, including proofs and additional material, can be found in the appendices.

2 Related Work

A considerable literature has been devoted to the design of dependently-typed programming languages. We do not give a survey of this vast field. Instead, we describe aspects of languages that are most related to DOL. In particular, we focus on languages that, in order to keep typechecking simple, restrict the power of dependent types by establishing a clear phase distinction between the language of computations and that of specifications.

Languages with full dependent types, such as Cayenne [2], Epigram [15], PIE [24], or Idris [4] do not make any distinction between the programming and
the specification languages. Cayenne, for example, makes type checking semi-decidable by forcing the typechecker to terminate within a number of prescribed steps. Epigram requires correctness proofs to be specified, since it builds on a tactic-driven proof engine, similar to that of Coq proof assistant [6]. Ynot [16] 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, whereby pre- and postconditions are captured in types. The idea has some similarities with input-output types from DOL. However, Ynot remains a tactic language, requiring programmers to construct proofs for their code.

An alternative approach makes a clear separation between the language used for computations and that used for reasoning about properties. Xi and Pfennig’s DML [30] introduces a restricted form of dependent types, extending a real functional programming language with index expressions that capture many program invariants in data structures and detect related errors at compile time. Type dependencies are allowed only on special terms of index sorts, not on terms of arbitrary types, reducing type checking to a constraint satisfaction problem on terms belonging to index sorts, which remains decidable. A particular application of the proposed type system is the elimination of the runtime array bound checking. Their work relates closely to Zenger’s indexed types [32].

Xanadu [27], a language with a C-like syntax, was later designed to demonstrate the benefits of combining imperative programming with dependent types, and, after that, came ATS [28], providing support to several programming paradigms, such as functional, imperative, concurrent and modular. DOL is imperative and, in this respect, it relates to Xanadu and ATS. However, in DOL, we introduce a dependently-typed discipline in a realistic class-based language, exhibiting a rigid class structure and membership, absent from other approaches, and including object-oriented programming features using a Java-like approach.

Ωmega [22] and Liquid Types [21] 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 [14] is a type-safe extension of the C programming language, combining static analysis and run-time checks. As DML, it offers domain-specific indexed types, but for the purpose of safe multi-threading and memory management. For example, a pointer can be annotated with the number of elements it points to in order to statically prevent buffer overflows. ConCoqtion [12] also reveals a strong separation between the computation language (which may include effects) and the specification language (which does not). In ConCoqtion, the language of computations is based on OCaml, using Coq as the specification language capturing correctness properties in types.

Flanagan et al. [11] extend to objects ideas from the hybrid type checking strategy for the pure functional lambda-calculus [10]. The proposed language uses an expressive type system of refinement types which is undecidable. The hybrid strategy is presented as an alternative to the problem of decidability. The idea is to catch statically as many type errors as possible, using dynamic checks when the typechecker cannot determine type safety. The language also offers
final and mutable objects and imperative method update, but the language is object-based, as opposed to the class-based style of DOL.

Another reference is Dependent JavaScript (DJS) [5], 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 and aliasing relationships, in combination with refinement types. The result is an increase in the language expressiveness through type annotations inside JavaScript comments that are able to account for side-effects. DOL is a class-based, static language, whereas JavaScript is a prototyped-based, dynamic language. However, like DJS, our language supports imperative updates combined with dependent types. DFJ adopts the Alias Types [23] mechanism by maintaining a heap, mapping locations to values (instead of types), in order to track changes originated by imperative updates. The solution proposed in DOL uses linearity in combination with more expressive types to control aliasing.

X10’s constrained types [18] can also be considered a form of dependent types. These types are designed around the notion of constraints on the immutable state of objects. Classes are parameterised by properties, which are treated as final public instance fields, and can be constrained to give methods more precise types. Even though there is a clear separation between compile time and runtime execution, properties that constrain types are directly embedded in computations, which has obvious advantages in terms of simplicity: the programmer can use the same variable in computation and to specify types, eliminating the need of code duplication required in approaches with strong phase distinction, such as that of DML and also of DOL. At compile time, queries are generated to the constraint solver engine. In theory, types can be constrained by any property. In practice, however, constraint restrictions are imposed in terms of the underlying constraint solver theory. X10’s approach is very appealing, and the framework is powerful. However, to our knowledge, support for reasoning about mutable objects and effects has not yet been provided.

Another related system, but with a different formal approach, is the Extended Static Checker tool (ESC/Java) [9] that uses JML for expressing specifications and automatic theorem proving techniques to detect many kinds of errors. While the tool is useful in practical programming, it is not intended to be sound or complete, but only to check specifications and give warnings of inconsistencies.

3 Motivating Examples

In this section, we motivate dependent mutable object types using the running example of the bank account class from Section 1. To illustrate the main features of DOL, we define a subclass CheckingAccount derived from Account. Then, we take advantage of the notion of subtyping and inheritance in DOL to present an inductively-defined list of transactions, and we add it to another subclass TransactionAccount that stores the account’s transaction history.
class Account(b:integer) ⇒ 
  init :: Account 0 
  deposit :: m:natural ⇒ Account b ↝ Account (b+m), Integer m 
  withdraw :: m:{x:natural | x ≤ b} ⇒ Account b ↝ Account (b−m), Integer m 
  getBalance :: Account b, x:integer ⇒ Integer x ⇝ Integer b 
  balance :: Integer b 
  init () = balance := 0 
  deposit(amount) = balance.add(amount) 
  withdraw(amount) = balance.sub(amount) 
  getBalance(bal) = bal := 0; bal.add(balance)

Fig. 2. The bank account

3.1 A Bank Account with Dependent Types

In DOL, the dependent function type is written \( \Delta \Rightarrow T \), where \( T \) may refer to (hence depend on) a context \( \Delta \) of typed indices of the form \((x_1: I_1, \ldots, x_n: I_n)\).

Following DML [26,31], index types \( I_1, \ldots, I_n \) are drawn from the particular constraint domain in use. For this paper, we fix an integer domain, restricting predicates to the decidable theory of linear inequalities, cf. [27]. The key idea is that if constraints generated during typechecking fall within a decidable theory, and can be solved by a constraint solver, typechecking can be proved to be decidable. We leave for future work other index term plugins, which may easily be incorporated in DOL, possibly at the cost of typechecking decidability.

In DOL, every class defines a type that acts as an “interface”, listing the methods that are available on a particular instance of the class. For example, the Account class type in Figure 2 specifies the four available methods, as well as a single instance attribute named balance. By convention, init (line 2) is the name of the method immediately called after object creation. Its body is the first piece of code executed to initialize the new object; hence it is the closest we have to a constructor.

Method signatures specify how arguments are allowed to change. A type \( T_1 \rightsquigarrow T_2 \) reads “an object type \( T_1 \) (at method entry) becomes \( T_2 \) (at method exit)”. To simplify, we often refer to \( T_1 \) as an object’s input type and \( T_2 \) as its output type. As mentioned in Section 1, the first type in every method signature, including init, is always the type of the current instance, by convention named this. In the init method, this refers to the newly created object, and, for this reason, the constructor signature is a special case where only the output type is specified. In the other methods, this refers to the object calling the method. Also recall that, in the examples, \((\rightsquigarrow)\) may be omitted when the input and output types are the same.

In DOL, we can easily distinguish accessor and mutator methods by their signature. The deposit and withdraw methods are mutators, since their signatures (lines 3–6) precisely describe the effect of the operation on the state of the caller.
1 class Integer (i: integer) ⇒
2 init :: Integer 0
3 add :: j: integer ⇒ Integer i ↦ Integer (i+j), Integer j
4 sub :: j: integer ⇒ Integer i ↦ Integer (i−j), Integer j
5 inc :: Integer i ↦ Integer (i+1)
6 dec :: Integer i ↦ Integer (i−1)
7 copy :: Integer i, x: integer ⇒ Integer x ↦ Integer i
8 // other signatures omitted

Fig. 3. The Integer interface

On the other hand, the getBalance method (line 7) is an accessor, since it does not act on the current instance (note the absence of the wavy arrow in the type of this). As previously mentioned, DOL does not support value-returning methods. Instead, all methods are state-modifying methods, in a sense like Java’s void methods. Hence, when entering getBalance, the type of the parameter bal is Integer x for some x of type integer, becoming Integer b at method exit when it holds the account balance. To illustrate how the index types evolve, consider the following code that creates an account and calls some methods on it:

```dol
var val := new Integer() // val : Integer 0
var myAccount := new Account(); // val: Integer 0, myAccount: Account 0
myAccount.deposit(620); // val: Integer 0, myAccount: Account 620
myAccount.withdraw(70); // val: Integer 0, myAccount: Account 550
myAccount.withdraw(65); // val: Integer 0, myAccount: Account 485
myAccount.getBalance(val) // val : Integer 485, myAccount: Account 485
```

For the sake of simplicity, DOL provides native classes for basic types, such as Integer. Unlike Java, DOL does not distinguish between primitive values and objects. Primitive values are objects, and programmers can call methods on these objects exactly as they would call the deposit method on an account instance. Figure 3 presents some of the available methods on the Integer class. In the examples, we use some abbreviations. Technically, the program constant 0, distinct from the index value 0, is syntactic sugar for the object reference returned by new Integer(), whereas the number 3 is short for a reference t such that t := new Integer(); t.inc(); t.inc(); t.inc() .

### 3.2 A Subclass with Dependent Types

Dependent types are useful even in the presence of subclasses. Like Java, DOL supports single class inheritance, and it provides a supertype Empty, a concrete class which has no fields or methods. In the examples, the programmer may omit the extends Empty declaration.

The dependent CheckingAccount class, defined in Figure 4, extends the Account class by setting an overdraft limit. To accomplish this, the class declares an additional index parameter n of type natural (line 1), which is the argument of
overdraftLimit type (line 6). The declaration `extends Account b` ensures that the supertype is well-formed. Let $\Delta \equiv b: \text{integer}, n: \text{natural}, b': \{x: \text{integer} \mid x \geq -n\}$, containing the index variables from the super and subclass. The typechecker can prove the subtype relation $\Delta \vdash \text{CheckingAccount} (n,b') <: \text{Account} b$, which involves checking the semantically defined judgement $\Delta \models b' \leq b$. In the decidable domain of linear inequalities, this constraint generated during typechecking can be solved automatically by a constraint solver.

Like Java, the `CheckingAccount` class inherits from the superclass all the fields and methods that are not overridden. The `CheckingAccount` defines a constructor that takes an argument of type `Integer m`, where `m: \text{natural}` (line 2), and makes a copy in order to initialise the `overdraftLimit` field (line 9). The subclass redefines the `withdraw` method (lines 3–5) so as to maintain the overdraft limit invariant as specified in the class. In DOL, covariant overriding is allowed, i.e. the signature of a method is a subtype of the signature of the corresponding method in the superclass. Let $\Delta \equiv \Delta_1, \Delta_2$ where $\Delta_1 \equiv b: \text{integer}, m: \{x: \text{natural} \mid x \leq b\}$ is the context from the `withdraw` method in the `Account` class, and $\Delta_2 \equiv n: \text{natural}, b': \{x: \text{integer} \mid x \geq -n\}, m': \{x: \text{natural} \mid x \leq b'+n\}$ is the context of the overriding method in the `CheckingAccount` class. The typechecker needs to verify the subtyping relations $\Delta \vdash \text{CheckingAccount} (n,b') <: \text{Account} b$ and $\Delta \vdash \text{Integer} m' <: \text{Integer} m$, which again involves proving $\Delta \models b' \leq b$.

Classes whose $\Delta$ is empty define freely shared, concrete types, as opposed to type families. Although orthogonal to the problem of using dependent types to more accurately reason about state, a special aliasing mechanism is introduced for types governed by indices. For example, the assignment `overdraftLimit := limit` in the `CheckingAccount` class may have unclear results unless the `limit` parameter is copied, as in the example (line 9) or is given the special type `Empty` at method exit. Having identified assignment and parameter passing as the potential sources of aliasing problems in DOL, we deal with the former by adopting a simple control mechanism in the formal language, which treats dependently-typed objects linearly. We do not employ the same mechanism to deal with parameter passing, because that would mean “emptying” an object passed as
class Node ⇒ {}

class Nil extends Node ⇒
init :: Nil
init () = {}

class Cons extends Node ⇒
init :: Cons, Node, Transaction
getValue :: Cons, Transaction
setValue :: Cons, Transaction
getNext :: Cons, Node
setNext :: Cons, Node
value :: Transaction
next :: Node

// implementation omitted

class SList(n:natural) ⇒
init :: SList 0
addFirst :: SList n <--> SList (n+1), Transaction
removeFirst :: SList (n+1) <--> SList n
get :: i:{x:natural | x<n} ⇒ SList n, Integer i, Transaction
getSize :: SList n, Integer 0 <--> Integer n
filter :: SList n, SList 0 <--> (m:{x:natural | x ≤ n} ⇒ SList m), Predicate
head :: Node
count :: Integer n

// remaining signatures and implementation omitted

Fig. 5. A dependently-typed singly linked list

an argument to a method. Instead, we take advantage of the call-by-reference strategy together with input-output types to track how the type evolves.

3.3 Singly and Doubly Linked Lists with Dependent Types

In order to define a TransactionAccount class that extends Account, we introduce in Figure 5 a singly linked list defined inductively in terms of the Nil and Cons classes, derived from the Node abstract class. In DOL, a class without an init method is like an abstract class in Java, since it cannot be instantiated. As null is not a value in DOL, we use a functional style approach, defining inductive data types. The list is defined using field head of type Node, denoting all possible forms of nodes. An empty list is one in which head is an instance of Nil, while in a non empty list head is an instance of Cons, containing two fields: a reference to a transaction (whose class we omit) and another reference to the next node.

The SList class introduces a parameter n of type natural to track its size. It declares safe methods for adding and removing a transaction object from the list, as well for accessing the nth transaction and obtaining the list size. The filter signature (line 20) provides an example of an existential type. The method takes an empty list as argument and the result is a list that contains all the transactions that satisfy the given Predicate. The resulting list has some
class TransactionAccount(n: natural, b: integer) extends Account b ⇒
init :: TransactionAccount (0,0)
transactions :: SList n
init () = super(); transactions := new SList()
// remaining signatures and implementation omitted

Fig. 6. A subclass tracking the account transaction history

class DCons(value: Transaction,
next: {n:Node | n instanceof DCons ⇒ next.prev = n}
prev: {n:Node | n instanceof DCons ⇒ prev.prev = n}) ⇒ ...

Fig. 7. A node for a doubly linked list

unknown size; the only condition that can be enforced by the filter type is that its size must smaller or equal to the current list size.

We are now in a position to define the type of TransactionAccount in Figure 6. The class simply declares an index variable n of type natural to keep track of the transaction list size, and an additional transactions field which keeps the history of all the deposits and withdrawals in the account.

Instead of being restricted to a domain of linear inequalities over the integers, as in the examples above, we could integrate in DOL a richer domain to express more expressive abstractions such as the ones proposed in Figure 7. The DCons class represents a node that can be used to construct doubly linked lists (possibly with self-referencing sentinel nodes). For this example, we would need a domain where we could express instanceof, field access, reference equality (and logic). Although decidability of type checking for this kind of non-trivial properties is problematic, we could adopt a hybrid approach (cf. [10,11]), relying on runtime checks for any remaining properties that could not be statically verified.

4 Syntax

In this section, we define the language syntax, which models a subset of features usually present in object-oriented programming languages, including class definitions, inheritance, field lookup and update and method calls. To constrain an object type and to reason about state modifications, we augment types with dependencies on values and method signatures with a relationship between input and output types. As in DML [26,29,31], we follow a syntax-split approach, restricting dependencies to terms of a specific index language that cannot be used in computations.

We distinguish between the user’s syntax in Figure 8, available to the programmer, and the runtime syntax in Figure 9, used only in the operational semantics and type system. As usual, a more practical syntax is used in the
examples in Section 3. To simplify the formal language, we introduce a number of restrictions on the syntax of specific constructs.

- Except for the constructor, every method declares exactly one parameter. It is easy to generalize and allow an arbitrary number of parameters, at the expense of increased verbosity in the typing rules. For simplicity sake, the constructor takes one parameter for each field, including all the inherited ones.
- A method call is available only on references (fields and parameters). A call on an arbitrary term could easily be derived, but it would complicate the typing rules.
- A method argument is always a reference. This is because a call may change the argument type. We believe it does not affect the language expressivity, since we can assign a term to a field, and then pass it to the method.

With respect to the examples in Section 3, we inline the call to super in init methods.

4.1 User’s Syntax

**Programs and classes.** A program \( P \) is given by a collection of class declarations \( L_1 \) through \( L_n \). A class declaration \( \text{class } C : \Pi \Delta \triangleleft C.\bar{x}.T \text{ is } M \) introduces a class named \( C \) that extends \((\triangleleft)\) a superclass of type \( D\bar{x} \), where \( \bar{x} \) is declared in \( \Delta \). A class has a type \( T \), and a class body, composed of methods \( M \). These take
the form of a record value \( \{ K, l_k(y_k) = t_k \}_{k \in 1...n} \), composed of a constructor \( K \) that assigns initial values to fields, and \( n \) members, identified by labels \( l_1 \) through \( l_n \). Both in records and record types, \( n \) is allowed to be 0, in which case the range \( 1 \ldots n \) is empty. In the examples, we write class \( C \) is \( \{ \} \) for an abstract class with no fields or methods. The class type \( T \) is expected to be a record type, containing types of fields and of methods defined in \( M \). We assume that class identifiers in a program and member names are all distinct. However, we do not require the sets of names to be disjoint from one class to another.

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

- A class name \( C \) is a nominal object type induced by classes, commonly found in most mainstream object-oriented languages. It is simple and easy to use, but it lacks the flexibility of a structural type. We combine the two, in the style of several other languages with advanced typing features [20]. In DOL, we use the nominal type as an alias for its structural representation as a record type.
- A record type \( \{ l_k : T_k \}_{k \in 1...n} \) exposes the class members: labels \( l_1 \) through \( l_n \), together with types \( T_1 \) through \( T_n \), form field and method types. If no members are defined in the class body, \( n \) is allowed to be 0. A class declaration combines a nominal and a structural type as follows:

\[
\text{class } C : \Pi \Delta \triangleleft D \bar{x}. \{ l_k : T_k \}_{k \in 1...n} \text{ is } \ldots
\]

As mentioned in Section 3, object types are inherently recursive, therefore \( C \) may occur in any \( T_j \) for \( 1 \leq j \leq n \). The class name provides for the recursion fixed point.
- A type family \( \Pi x : I . T \) is a type that maps elements of index type \( I \) to elements of the main type \( T \), where \( x \) may occur free in \( T \). It can be used to build up class and method types.
- A type application \( T i \) instantiates a type family. We use it to give types to objects. In contrast to ordinary types, a dependent object type can vary according to the index term supplied. As explained in Section 3, \( \text{SList } 0 \) is the type of an empty list and \( \text{SList } (n-1) \), with \( n \) ranging over all positive numbers, is the type we give to a list of type \( \text{SList } n \) after an item is removed.
- An existential type \( \Sigma x : I . T \) classifies objects of type \( T \) where \( x \) of index type \( I \) represents some unknown value in \( T \).
- A pair of types \( T \times T \) is used in a method signature. The first type is the type of this, implicitly passed to the method, while the second one is the type of the method’s parameters.
- A parameter type \( T \rightarrow T \) defines a relationship between an input and a possibly different output type, corresponding to the view of an object at method entry and exit.
Terms. With respect to ordinary terms, we have (in order of appearance) references \( s \), representing identifiers \( y \) and the selection \( y.l \) used for field access, assignment to fields, object creation and method calls in the form of term application, the conditional, the loop and the sequential term composition. Identifiers include the keyword \texttt{this} that specifies the current instance. In the examples, field accesses may omit the prefix \texttt{this}; the compiler can insert it when needed. In the runtime syntax, identifiers further include \( o \), representing an object identifier. We always use the letter \( y \) to distinguish object identifiers from index identifiers \( x \), used only in types.

Index constructs. We present only a subset of the possible index constructs. Index terms \( i \) include (in order of appearance) variables, integer literals, addition of index terms, and a function that returns the greater of two index terms. Index types \( I \) comprise the integer type and the refinement type of the form \( \{ x : I \mid p \} \), constrained by a predicate \( p \). In our case, these are conjunctions of inequations. Sections 1 and 3 introduce some examples of refinement types.

Context \( \Delta \) maps index variables to index types. In the type system, \( \Delta \) is used as a type environment of indices. The standard set notation \( x \in \Delta \) is used to refer to an index variable defined in \( \Delta \). We write \( \Delta, x : I \) only if \( x \notin \Delta \). We use the same notation and convention for heap and other type environment membership.

4.2 Runtime Syntax

Figure 9 defines additional syntax that is required for the formal system, but is inaccessible to the programmer. The only values in our language are object identifiers \( o \). We distinguish the \texttt{oEmpty} of type \texttt{Empty} as a base object. A heap \( h \) is a mapping from object identifiers \( o \) to records \( R \). The heap produced by the operation \( h, o = R \) contains a new mapping from \( o \) to \( R \). This operation is only defined if \( o \notin h \). Object records are instances of classes and are represented by \( C \{ l_k = o_k^{k \in 1 \ldots m} \} \), a fixed class name followed by a mutable record value of fields.
Fig. 10. Auxiliary definitions

For the typing and reduction rules, we need a few auxiliary functions, given in Figure 10. Function \( \text{ctype} \) looks up a class type, and \( \text{override} \) checks if a method type can be overridden. \( T_1 + T_2 \) requires that both \( T_1 \) and \( T_2 \) are record types, and \( l:T \in (T_1 + T_2) \) means either \( l:T \in T_1 \), or \( l:T \in T_2 \) when \( l:T \not\in T_1 \), with \( l \neq \text{init} \).

5 Reduction

The operational semantics is defined as a reduction relation on states \( S \), introduced earlier in Figure 9. These consist of two components: a heap and a term. \( E \) are evaluation contexts in the style of Wright and Felleisen [25]. Intuitively, an evaluation context is a term with a hole \( \_[\_] \) at the point where the next reduction step must take place in a call-by-value evaluation order; \( E[t] \) is the evaluation context obtained by replacing the hole in \( E \) by \( t \).

The main consequence of the phase distinction design decision is that there is no need to evaluate index terms, since the compiler does not run programs that do not typecheck. This, together with progress and subject reduction, guarantees type soundness for our language.

In the reduction rules, we use some useful notations and conventions.

**Definition 1 (Operations on Heaps).**

- If \( R = C\{l_k = o_k \mid k \in 1 \ldots m\} \), then \( R.\text{class} \triangleq C \) and \( R.l_j \triangleq o_j \) for any \( 1 \leq j \leq m \). We also define \( R\{l_j \mapsto o\} \triangleq C\{l_k = o'_{k \in 1 \ldots n}\} \) where \( o'_{k} = o_{k} \) for \( k \neq j \) and \( o'_j = o \).
- If \( h = (h', o = R) \) and \( l \) is a field of \( R \), then \( h(o,l) \triangleq h(o).l \) and \( h\{o.l \mapsto o'\} \triangleq (h', o = R\{l \mapsto o'\}) \).

In order to access elements in class \( C \colon \Pi \Delta \triangleq D \alpha \cdot T \) is \( M \), we define some additional abbreviations. \( C.\text{fields} = \tilde{l} \) denotes the fields in class \( C \) and its superclass. We also write \( l(y) = t \in C \) to mean \( l(y) = t \in M \). Finally, if \( h(o) = C\{l_k = o_k \mid k \in 1 \ldots n\} \), then \( h(o).\text{class} \) means \( C \). As usual, we denote by \( t[y/x] \) the capture-avoiding substitution of \( o \) for the free occurrences of \( y \) in \( t \), defined in the standard way. The reduction rules in Figure 11 are straightforward.

The reduction of the following terms does not change the heap: R-FIELD simply reduces the field being accessed to the object’s value; R-CALL prepares
\[
S_1 \rightarrow S_2
\]

\[
(h, o.l) \rightarrow (h, h(o), l) \quad \text{(R-FIELD)}
\]

\[
\text{ctype}(h(o')\.\text{class}) = \{ k \in T_k^{1 \ldots n} \}
\]

\[
(h, o.l \Leftarrow o') \rightarrow (h\{o.l \mapsto o'\}, o'_\text{Empty}) \quad \text{(R-UnAssign)}
\]

\[
\text{ctype}(h(o', t').\.\text{class}) = \Pi \Delta T
\]

\[
(h, o.l \Leftarrow o'.t') \rightarrow (h\{o.l \mapsto h(o').t'\}{o'.t' \mapsto \text{Empty}}\}, o'_\text{Empty}) \quad \text{(R-LinAssign)}
\]

\[
C.\text{fields} = t \quad o' \text{ fresh} \quad h' = h, (o' = C\{t = a\})
\]

\[
(h, o.l \Leftarrow (\text{new} C)t) \rightarrow (h\{o.l \mapsto o'\}, o'_\text{Empty}) \quad \text{(R-NewAssign)}
\]

\[
l(y) = t \in h(o).\text{class}
\]

\[
(h, (o.l)o') \rightarrow (h, l^{o/y}_{t/\text{this}}) \quad \text{(R-Call)}
\]

\[
(h, \text{if } o = o \text{ then } t_1 \text{ else } t_2) \rightarrow (h, t_1) \quad \text{(R-IfTrue)}
\]

\[
(h, \text{if } o = o' \text{ then } t_1 \text{ else } t_2) \rightarrow (h, t_2) \quad \text{(R-IfFalse)}
\]

\[
(h, \text{while } o = o' \text{ do } t) \rightarrow (h, \text{if } o = o' \text{ then } (t; \text{while } o = o' \text{ do } t) \text{ else } o'_\text{Empty}) \quad \text{(R-While)}
\]

\[
(h, o; t) \rightarrow (h, t) \quad \text{(R-Seq)}
\]

\[
(h_1, t_1) \rightarrow (h_2, t_2) \quad (h_1, \mathcal{E}[t_1]) \rightarrow (h_2, \mathcal{E}[t_2]) \quad \text{(R-Context)}
\]

Fig. 11. Reduction rules

The method body \( t \) with two substitutions (the current object for \text{this} and the actual parameter for the formal parameter); \text{R-IfTrue} and \text{R-IfFalse} make each branch depend on the object comparison that controls the condition; the behaviour of the while loop is defined by rewriting it to a nested conditional term – if the test succeeds, an iteration will be performed, if it fails, the iteration is cancelled; \text{R-Seq} reduces the result to the second part of the term sequence, discarding the first part only after it has become a value; and \text{R-Context} is standard, defining which expression should be evaluated next.

We now present the reduction rules that affect the heap: \text{R-UnAssign} is the rule for assigning a value of an unrestricted class (not governed by a \( \Delta \)), updating the heap by replacing the field value with \( o' \); \text{R-LinAssign} evaluates assignment of a field of a dependent class – as \text{R-Assign}, it updates the heap by replacing the field with the record in \( h(o').t' \) and the right-hand field is replaced with \( \text{Empty}\{} \), returning the empty object as part of the linear treatment of objects; \text{R-NewAssign} adds to the heap a fresh object, with initialised fields, and updates the heap by replacing the field value with the new object.

16
Given a program composed of a collection of class declarations, we define class Main, with no dependencies, as the program starting point with a special method run, invoked to begin execution. We assume an initial empty heap defined as follows:

\[ h_0 \triangleq o_{\text{Empty}} = \text{Empty}\{\} \]

to which we add an object \( o \) of type Main. Note that since the Empty class provides no side-effecting operations, the DOL program reuses the unique reference \( o_{\text{Empty}} \). Intuitively, we can simplify and say that a program starts with 
\( \text{new Main().run()} \). According to the syntax defined, run should take a parameter \( y \); let it be \( o_{\text{Empty}} : \text{Empty} \). Formally, a program’s initial state is therefore

\[ (h_0, o = \text{Main}\{l_k = o_{\text{Empty}} \mid k \in 1 \ldots n\}, (o.\text{init}; (o.\text{run}) o_{\text{Empty}})) \]

6 Typing

In this section, we present the type system for DOL. We typecheck our language with respect to two initial type environments: \( \Delta \), containing bindings from index variables to index types, is the environment used to check type formation and subtyping, and \( \Gamma \), mapping object identifiers to ordinary types, is the standard, computational environment used to check ordinary terms. It is defined as follows:

\[ \Gamma ::= () \mid \Gamma, y : T \]

Ordering is important in index environment \( \Delta \), because of dependencies, and irrelevant in the standard environment \( \Gamma \). The main judgements consist of \( \Delta \vdash I \) and \( \Delta \vdash T \) for type formation, \( \Delta \vdash I < : J \) and \( \Delta \vdash T < : U \) for subtyping, and \( \Delta \vdash i : I \) and \( \Delta; \Gamma_1 \vdash t : T \vdash \Gamma_2 \) for term typing. The typing judgement for index terms is straightforward. The typing judgement for ordinary terms is the most involved, showing that \( t \) may 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 environment \( \Gamma_2 \). Note that \( \Delta \) does not vary, since index variables do not take part in evaluation. Following [31], we also assume in the typing rules the semantically defined judgement \( \Delta \models p \). Sometimes the letters \( U \) and \( J \) may also be used to range over ordinary and index types, respectively.

6.1 Index Binding and Substitution

The binding occurrences are the typed index variables \( x : I \) in the refinement type, and in the universal and existential types. We say that \( x \) occurs bound in \( p \) within the refinement type \( \{x : I \mid p\} \) and in \( T \) within \( \Pi x : I.T \) and \( \Sigma x : I.T \). To simplify the proofs, in particular to avoid having to rename bound variables in substitution, we assume that no variable can be both free and bound. We follow the Barendregt’s Variable Convention [3] whereby the names of bound variables
\[ \Delta \vdash T \]

\[
\frac{\text{class } C : \Pi \Delta \leq D x : T . M}{\Delta \vdash C} \quad \text{(WF-CLASS)}
\]

\[
\forall 1 \leq k \leq n \quad \frac{\Delta \vdash \{ k : T_k \}_{k \in 1 \ldots n}}{\Delta \vdash T} \quad \text{(WF-RECORD)}
\]

\[
\frac{\Delta \vdash T}{\Delta \vdash T \ < : H x : I . T} \quad \frac{\Delta \vdash i : I}{\Delta \vdash i} \quad \text{(WF-APP)}
\]

\[
\frac{\Delta \vdash I \quad \Delta, x : I \vdash T}{\Delta \vdash H x : I . T} \quad \frac{\Delta \vdash I \quad \Delta, x : I \vdash T}{\Delta \vdash \Sigma x : I . T} \quad \text{(WF-\(\Pi\)) \ (WF-\(\Sigma\))}
\]

\[
\frac{\Delta \vdash T_1 \quad \Delta \vdash T_2}{\Delta \vdash T_1 \rightsquigarrow T_2} \quad \text{(WF-\(\rightsquigarrow\))} \quad \frac{\Delta \vdash T_1 \quad \Delta \vdash T_2}{\Delta \vdash T_1 \times T_2} \quad \text{(WF-\(\times\))}
\]

Fig. 12. Rules for type formation.

must all be distinct from each other and from any other variables occurring free in declarations of the form \(x : I\).

We denote by \(i_1[^{i_2/x}]\) the capture-avoiding substitution of \(i_2\) for the free occurrences of \(x\) in \(i_1\). Index substitutions are defined inductively on the structure of index terms. For example, \((i_1 + i_2)^[^{i_3/x}]\) is defined as \(i_1[^{i_3/x}] + i_2[^{i_3/x}]\).

A single index substitution is extended pointwise to multiple substitution \(\theta\), which maps index variables to index terms, by defining \(i(\theta)^[^{i_3/x}]\) is defined as \(i[^{i_3/x}] \theta\). The judgement for deriving \(\theta\) is of the form \(\Delta \vdash \theta : \Delta_2\) and requires that \(\theta\) and \(\Delta_2\) have the same number of elements and that each substituent is well-formed in the environment. 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 \(T\). Details of the substitution rules can be found in Appendix A.

6.2 Well-Formedness

We write \(\Delta \vdash T\) if \(T\) is well-formed under index environment \(\Delta\). For this to happen, a type must be closed under \(\Delta\), assuming all the index types are well-formed. This requirement is formalized in the rules given in Figure 12. We include in Appendix A the standard rules for the well-formedness of index types, propositions, and contexts. As usual in dependent type theory, the judgements are mutually inductively defined.

WF-CLASS defines a legal class name as one for which there is a class declaration. WF-RECORD checks that each member type is well-formed. WF-APP checks the instantiation of a type family. WF-\(\Pi\) and WF-\(\Sigma\) invoke the well-formedness of program types and index types. WF-\(\rightsquigarrow\) and WF-\(\times\) are standard, checking that each type component is well-formed.

Checking index terms can be done in a straightforward way following the rules, adapted from Xi [26,31]. The rules can be found in Appendix A.

18
6.3 Subtyping

The subtyping relation in DOL is given in Figure 13. Our rules are formulated based on the subtyping rules from Aspinall and Compagnoni [1] and Dunfield and Pfenning [7,8] proved to be decidable. Reflexivity and transitivity are explicitly expressed by rules S-Refl and S-Trans. In rules S-Subclass, S-β, S-ΠL, and S-App, we assume that $\mid i \leq j$ in the constraint domain is decidable. In rules S-IIl and S-ΣR, the index term $i$ is “guessed”, closely following Dunfield’s approach [7] which relies on introducing and solving existentially quantified variables.

S-Subclass defines a relation between a class and its superclass as one where the class may define more index parameters than its superclass. S-ClassL/R allow a nominal type $C$ to be used in place of its structural type, given by the function $\text{ctype}(C)$. The $\beta$-reduction for types is included via rules S-β and S-App. In S-IIl/R, the left rule instantiates the index variable $x$ to $i$ in the subtype, while the right rule relates two types $T$ and $U$ regardless of an index variable $b$ appearing free in $U$ (see an example below). The remaining rules are congruences.

Using these rules, we can build a derivation tree for checking if withdraw in the CheckingAccount class from Section 3 is a valid method overriding of the
\[ I_1 \triangleq \{ x: \text{natural} \mid x \leq b \}, I_2 \triangleq \{ x: \text{integer} \mid x \geq -n \}, I_3 \triangleq \{ x: \text{natural} \mid x \leq b + n \} \]

\[ \Delta' \triangleq n: \text{natural}, b_C : I_2, m_C : I_3 \]

\[ \Delta \triangleq \Delta', b_A : \text{integer}, m_A : I_1 \]

\[ W_A \triangleq \Pi b_A : \text{integer}. \Pi m_A : I_1. b_A \leadsto A(b_a - m_A) \times Zm_A \leadsto Zm_A \]

\[ W_C \triangleq \Pi n : \text{natural}. \Pi b_C : I_2. \Pi m_C : I_3. (Cn)b_C \leadsto (Cn)(b_C - m_C) \times Zm_C \leadsto Zm_C \]

(2) \[ \frac{\text{class } C : \Pi \Delta' \triangleq \text{Ab}_C.T \text{is } M}{\Delta \vdash \text{Cn} < : A} \quad \text{(S-SUBCLASS)} \]

(1) \[ \frac{\Delta \vdash b_C - m_C \leq b_A - m_A}{\Delta \vdash (\text{Cn})(b_C - m_C) < : A(b_a - m_A)} \quad \text{(S-CLASS)} \]

\[ \frac{\Delta \vdash (\text{Cn})b_C \leadsto (\text{Cn})(b_C - m_C) < : \text{Ab}_A \leadsto A(b_a - m_A)}{\Delta \vdash W_C < : W_A} \quad \text{(S-IRR/L\*)} \]

Fig. 14. Subtyping derivation

\[ \text{class } C : \Pi \Delta \triangleq D \bar{x}. T \text{is } M \]

\[ \Delta = \bar{x} : \bar{I}, \bar{x} : \bar{J} \quad \Delta \vdash \text{(ctype}_A(D)[\bar{C}][\bar{I}])\bar{x} \downarrow \beta U \]

\[ \text{ctype}_A(\text{Empty}) = \{ \text{init : Empty} \} \]

\[ \text{ctype}_A(C) = \Pi \Delta.(T + U) \]

Fig. 15. Algorithmic auxiliary definitions

method with same name in class Account. For space reasons, we use the following abbreviations: \( W_A \) for the withdraw type in the Account class and \( W_C \) for the type of the corresponding method in the CheckingAccount class where \( A, C, Z \) range over the Account, CheckingAccount and Integer class names, respectively. In Figure 14, we show the subtyping relation \( \Delta \vdash W_C < : W_A \) using S-SUBCLASS, S-CLASS, S-IRR, S-
\( \sim \), S-\( \times \) and several applications of S-IRR/L.

The rules in Figure 13 are nondeterministic. Even though we cannot eliminate the nondeterminism arising from S-IRR/L and S-\( \Sigma \)R/L, we can eliminate the nondeterminism of S-TRANS, replacing the declarative subtyping system by an equivalent, syntax directed one. This system defines a relation \( \Delta \vdash T < : A U \), equivalent to \( \Delta \vdash T < : U \) (details are provided in Appendix C). In the derivation in 14, we do not know which rule to apply first S-\( \Pi \)L or S-\( \Pi \)L. In the algorithmic system in Figure 16, we write \( U^o \) in SA-\( \Pi \)L to denote a type \( U \) that is not a \( \Pi \). Similarly, \( T^o \) in SA-\( \Sigma \)R denotes a type \( T \) that is not a \( \Sigma \). As usual, the algorithmic system drops S-REFL and S-TRANS; additionally it drops S-CLASSL/R and S-\( \beta \), adding S-CLASS and S-APPL/R in order to grasp the \( \beta \)-reduction and
the structure of the subtyping relation in restricted occurrences of applications. Both S-ClassL/R use a convergence judgement and an algorithmic function \( \text{ctype}_A \) (Figure 15) for the reflexive transitive closure of \( \rightarrow^*_\beta \) defined as follows:

\[
\Delta \vdash \text{ctype}_A(C) \downarrow_\beta U \iff (\Pi x : I.T)i \rightarrow^*_\beta T[i/x]
\]

6.4 Typing Rules

We now define additional notation that we use in the typing rules.

**Definition 2 (Operations on Type Enviroments).**

- If \( \Gamma = \Gamma_1, y : T_1, \Gamma_2 \) then \( \Gamma(y) \triangleq T_1 \) and \( \Gamma \{ y \mapsto T_2 \} \triangleq \Gamma_1, y : T_2, \Gamma_2 \).
- If \( \Gamma = \Gamma', y : \{ l_k : T_k^{k \in [1\ldots n]} \} \), then \( \Gamma(y,l_j) \triangleq T_j \) and \( \Gamma \{ y,l_j \mapsto T \} \triangleq \Gamma', y : \{ l_k : T_k^{k \in [1\ldots n]} \} \) where \( T_k' = T_k \) for \( k \neq j \) and \( T_j' = T \) for any \( 1 \leq j \leq n \).
The following is used to interpret access and update to components of a class type.

**Definition 3 (Operations on Record Types).** Assume a class declaration of the form
\[ \text{class } C : \Pi \Delta \triangleleft D \bar{x}.T \text{ is } M \]
where \( \text{ctype}(C) = \Pi \Delta. T' \) and \( T' \triangleq \{ l_k : T_k \}_{k=1...m+n} \) and \( l_j \) is a method of type \( T_j \triangleq \text{mtype}(C, l_j) \) for all \( 1 \leq j \leq m \) and \( l_{j'} \) is a field for all \( m+1 \leq j' \leq m \). Then, \( C.l_j \triangleq T'.l_j \) and \( T'.l_j \triangleq \text{mtype}(C, l_j) \). We also define \( T'\{l_j, \rightarrow U\} \triangleq \{ l_k : T_k \}_{k=1...m+n} \) where \( T'_k = T_k \) for \( k \neq j' \) and \( T'_{j'} = U \).

**Typing programs.** We introduce three judgement forms in Figure 17 for checking programs, classes, and methods. T-PROGRAM checks a set of classes by calling T-Class, or T-AbstractClass when an init method is not provided. These two rules check if the class and the superclass types and each method declaration are well-formed. T-Class also shows how to initialise fields of an instance of \( C \); for simplicity sake, a subclass initialises the instance and the superclass fields from the parameters \( \bar{y} \). T-METHOD checks a method body, ensuring consistency between the input and output types declared in the method signature and the types bound to variables in the environment immediately before and after the method body is checked.

**Typing terms in the user’s syntax.** The typing judgement for terms is of the form \( \Delta; \Gamma_1 \vdash t : T \vdash \Gamma_2 \). The meaning of the parameters is as follows:
\[
\Delta; \Gamma \vdash t : T \rightarrow \Gamma_2
\]
\[
\begin{array}{ll}
\Delta \vdash \Gamma & \frac{y : T \in \Gamma}{\Delta; \Gamma \vdash y : T \rightarrow \Gamma} \quad (T-\text{VAR}) \\
\Delta; \Gamma \vdash y : T \rightarrow \Gamma & \frac{\Delta; \Gamma \vdash y : T \rightarrow \Gamma}{\Delta; \Gamma \vdash y.l : T.l \rightarrow \Gamma} \quad (T-\text{FIELD})
\end{array}
\]

\[
\begin{array}{ll}
\Delta; \Gamma \vdash s : C \rightarrow \Gamma & \frac{\Delta; \Gamma \vdash y.l : C \rightarrow \Gamma}{\Delta; \Gamma \vdash y.l := s : \text{Empty} \rightarrow \Gamma} \quad (T-\text{UNASSIGN})
\end{array}
\]

\[
\begin{array}{ll}
\Delta; \Gamma \vdash s : T^1 \rightarrow \Gamma & \frac{\Delta; \Gamma \vdash y.l : T i \rightarrow \Gamma}{\Delta; \Gamma \vdash y.l := s : \text{Empty} \rightarrow \Gamma[y.l \mapsto T^1 \{s \mapsto \text{Empty}\}] \quad (T-\text{LINASSIGN})
\end{array}
\]

\[
\Delta; \Gamma \vdash y.l : (\text{new } C) \bar{s} : \text{Empty} \rightarrow \Gamma[s \mapsto \bar{U}' \theta \{y.l \mapsto T'\}] \quad (T-\text{NEWASSIGN})
\]

\[
\Delta; \Gamma \vdash s_1 : T \rightarrow \Gamma & \frac{T.l = \Pi \Delta'.(T_1 \leadsto T_2) \times (U_1 \leadsto U_2)}{\Delta; \Gamma \vdash \theta : \Delta' \rightarrow T <: T_1 \theta} \quad (T-\text{CALL})
\]

\[
\begin{array}{ll}
\Delta; \Gamma \vdash (s_1.l)s_2 : \text{Empty} \rightarrow \Gamma[s_2 \mapsto \bar{U}_2 \theta \{s_1 \mapsto T_2 \theta\}] & \frac{\Delta; \Gamma \vdash s_2 : U_1 \theta \rightarrow \Gamma}{\Delta; \Gamma \vdash \theta : \Delta' \rightarrow T <: T_1 \theta}
\end{array}
\]

\[
\Delta; \Gamma \vdash s_1, s_2 : T_1 \rightarrow \Gamma_1 & \frac{\Delta; \Gamma \vdash t_1, t_2 : T_2 \rightarrow \Gamma_2}{\Delta; \Gamma \vdash \text{if } s_1 = s_2 \text{ then } t_1 \text{ else } t_2 : T_2 \rightarrow \Gamma_2} \quad (T-\text{IF})
\]

\[
\Delta; \Gamma \vdash s_1, s_2 : T_1 \rightarrow \Gamma_1 & \frac{\Delta; \Gamma \vdash t : T_2 \rightarrow \Gamma_2}{\Delta; \Gamma \vdash \text{while } s_1 = s_2 \text{ do } t : \text{Empty} \rightarrow \Gamma_1} \quad (T-\text{WHILE})
\]

\[
\Delta; \Gamma \vdash t_1 : T_1 \rightarrow \Gamma_2 & \frac{\Delta; \Gamma \vdash t_2 : T_2 \rightarrow \Gamma_3}{\Delta; \Gamma \vdash \text{Seq } t_1, t_2 : T_2 \rightarrow \Gamma_3} \quad (T-\text{SEQ})
\]

\[
\Delta; \Gamma \vdash t : T_2 \rightarrow \Gamma_2 & \frac{\Delta; \Gamma \vdash t_1 : T_1 \rightarrow \Gamma_1}{\Delta; \Gamma \vdash \text{Sub } t_1 \rightarrow t_2 : \text{Empty} \rightarrow \Gamma_1} \quad (T-\text{SUB})
\]

Fig. 18. Rules for typing terms

- The domain of $\Delta$, which maps index variables to index types, includes all the free variables in the codomain of $\Gamma_1$ and $\Gamma_2$ and in the type $T$ derived from the judgement.
- $\Gamma_2$ on the right-hand side of the judgement shows the change, if any, that $t$ causes in the initial type environment $\Gamma_1$.

The typing rules for terms are in Figure 18 (as specified by the syntax in Figure 8). T-VAR is used to access an identifier, either this or a parameter, while T-FIELD is the rule for accessing a class instance. We define two separate rules for field update. T-UNASSIGN types assignment of references having concrete types $C$. This distinction is needed as part of the linear control of objects, with T-LINASSIGN updating the final type environment by giving type Empty to the right-hand side reference. T-NEWASSIGN types assignment of a new object, requiring a constructor to be defined in the class type. Substitution $\theta$ maps index terms in the class index type environment to index variables defined in the
\[ \Delta; \Gamma \vdash \Delta; I_1 \vdash h : T \vdash I_2 \]

\[ \Delta; I_1 \vdash h \quad \Delta; I_1 \vdash t : T \vdash I_2 \]

\[ \Delta; I_1 \vdash (h, t) : T \vdash I_2 \]  (T-STATE)

\[ \Delta; \Gamma \vdash h \]

\[ \Delta; () \vdash () \]  (T-HEAPEMPTY)

\[ \Delta; \Gamma \vdash h \quad \forall 1 \leq k \leq m \quad \Delta; \Gamma \vdash o_k : T_k \vdash \Gamma \]

\[ \Delta; \Gamma, o : \{ l_k : T_k^{k \in 1 \ldots m+n} \} \vdash h = C \{ l_k = o_k^{k \in 1 \ldots m} \} \]  (T-HEAPOBJECT)

Fig. 19. Typing rules for the runtime syntax

method’s \( \Delta' \). In the final type environment, parameters are given their output types and the left-hand reference is given the type of the newly created object.

T-CALL checks that the caller has a type \( T \) that allows the signature of a method \( l \) to be obtained. The caller type is further expected to be a subtype of \( T_1\theta \), with the substitution applied, and the formal parameter is expected to have type \( U_1\theta \) (by, possibly, zero or more applications of T-SUB). The final type environment is updated so as to contain both the caller and the formal parameter output types.

T-IF checks the two references being compared, requiring that both branches of the conditional term use the same initial environment \( \Gamma_1 \). Since only one of the branches can be executed, it also enforces that they both share the same final environment, \( \Gamma_2 \). T-WHILE is similar, yet simpler. T-SEQ checks the first sub-term and considers its possible effects within the type environment that checks the second one. T-SUB is the standard subsumption rule.

Typing terms in the runtime syntax. Figure 19 describes how the two parts of a runtime state, a heap and a runtime term (see Figure 9), are related by typing. T-STATE checks that the type environment reflects the content of the heap; this constraint is given by judgement \( \Delta; I_1 \vdash h \). Rule T-HEAPEMPTY types an empty heap, and T-HEAPOBJECT is the only rule that adds an object to \( \Gamma \), ensuring the crucial property that every object identifier in \( \Gamma \) has a type that agrees with its value in \( h \). The rule further guarantees that an environment \( \Gamma \), which contains \( o_k \) in its entries, cannot contain \( o \), the object being added to the heap.

7 Properties

Under the assumption that \( \models \) in the constraint domain is decidable, we can show:

Theorem 1 (Subtyping Equivalence). \( \Delta \vdash T <: U \) iff \( \Delta \vdash T <: \mathcal{A} U \).
Proof. We establish completeness and soundness in Lemmas 13 and Lemmas 18 (see Appendix C).

Theorem 2 (Decidability of Subtyping). $\Delta \vdash T <: U$ is decidable. Moreover, by the previous result, this extends for deciding $\Delta \vdash T <: U$.

Proof. For any rule in Figure 16, each premise is smaller than the conclusion.

In this section, we also state and prove the main result of the paper, type soundness via subject reduction properties and progress. Using standard techniques [25], we can prove that if a state $S_1$ can make a transition, then the resulting state $S_2$ is of the appropriate type, even though a computation step may be taken that changes types in the environment along the computations.

Theorem 3 (Subject Reduction). If a program is well-formed and $\Delta; \Gamma_1 \vdash S_1 : T \vdash \Gamma_2$ and $S_1 \rightsquigarrow S_2$, then there exists $\Gamma_3$ such that $\Delta; \Gamma_3 \vdash S_2 : T \vdash \Gamma_2$.

Proof (Sketch). In order to build this result, we need to prove a number of lemmas such as weakening, permutation, substitution and agreement between the heap and the type environment, as well as changes in the heap under reduction. The agreement lemma shows the correspondence between an object type and its representation as a record in the heap, and proves that this relationship prevents circular references from occurring in the heap. The heap update lemma states that, under the appropriate conditions, heap update preserves heap well-formedness. In particular, given $\Delta; \Gamma \vdash s : T' \vdash \Gamma$ and $\Delta; \Gamma \vdash o : T \vdash \Gamma$, if a reference $s$ is updated with respect to a well-formed heap $\Delta; \Gamma \vdash h$, then the resulting heap $\Delta; \Gamma\{s \mapsto T\} \vdash h\{s \mapsto o\}$ is well-formed. The proof also uses the replacement lemma, a subterm typing lemma in the style of Wright and Felleisen[25], as well as Definition 1 from Section 5 for operations on heaps and Definitions 2 and 3 from Section 6 for operations on type environments and record types. Full details of this proof can be found in Appendix C.

Theorem 4 (Progress). If $\Delta; \Gamma_1 \vdash (h_1, t_1) : T \vdash \Gamma_2$, then $t_1$ is a value or $(h_1, t_1) \rightsquigarrow (h_2, t_2)$.

Proof. By case analysis on the form of $t$, using the appropriate clause of the inversion lemma. Details are provided in the non-anonymous supplemental material submitted with this paper.

8 Conclusion and Future Work

In this paper, we present an object-oriented language with dependent types, designed to support the verification of mutable objects. We adopt from other work [30,31] a restricted form of dependent types where special terms in types describe values in a program as an approximation to decidable typechecking.

The first contribution of our work is the introduction of dependent types at the class level as a way to formally specify invariants describing the consistent
states of an object. We are able to complement the specifications provided by
the object-level invariant through method signatures that capture in parameter
input-output types the relationship between the state of an object immediately
before and after a method call. We view this as a way to formally encode pre- and
postconditions in types, and it is our second and main contribution to integrating
dependent types in object-oriented programming. Support for single inheritance
in classes with dependent types is also provided. The last contribution presented
in this paper is the definition of a practical typechecking algorithm based on the
phase separation approach, closely following other work on functional and im-
perative dependently-typed languages as mentioned in the related work (Section
2).

We have illustrated the language main features with several practical ex-
amples involving a class representing a bank account indexed by the current
balance, subclasses derived from the Account class, and an inductively-defined
list.

We have formalised the syntax, type system and operational semantics for our
object-oriented language with support for mutable objects. We have proved type
soundness through progress and subject reduction theorems. The language may
be called “pure” object-oriented in the sense that objects are the only possible
values, and clients can only interact with objects to change state using methods
available in the object’s class.

In future work, we plan to continue the development of DOL, namely by
providing an implementation. We are currently studying the integration in DOL
of richer index languages in domains of interest, such as the one presented in
Section 3 that can be used to construct dependently-typed doubly linked lists.
Finally, it would be interesting to use dependent types to explore the idea of
properties across several objects, following the suggestion by Parkinson [19] that
when considering aggregate structures the object invariant is an unneeded mecha-

nism for the verification of objects.

References

2. Augustsson, L.: Cayenne a language with dependent types. In: International Con-
4. Brady, E.: Idris, a general-purpose dependently typed programming language: De-
Press (2012)
University (2007), cMU-CS-07-129
8. Dunfield, J., Pfenning, F.: Type assignment for intersections and unions in call-
by-value languages. In: Foundations of Software Science and Computational Struc-
12. Fogarty, S., Pasalic, E., Siek, J.G., Taha, W.: Concoqtion: indexed types now! In:
14. Jim, T., Morrisett, J.G., Grossman, D., Hicks, M.W., Cheney, J., Wang, Y.: Cy-
dent types for imperative programs. In: International Conference on Functional
17. Norell, U.: Towards a practical programming language based on dependent type
University of Technology (2007)
18. Nystrom, N., Saraswat, V., Palsberg, J., Grothoff, C.: Constrained types for object-
on Aliasing, Confinement and Ownership. (2007)
Functional Programming (2007)
and Computation 115(1), 38–94 (1994)
University, Pittsburgh (1998)
28. Xi, H.: Applied type system: Extended abstract. In: Types for Proofs and Pro-
29. Xi, H.: Dependent ML: an approach to practical programming with dependent
30. Xi, H., Pfenning, F.: Eliminating array bound checking through dependent types.
In: ACM SIGPLAN Conference on Programming Language Design and Implement-
A.1 Well-Formedness

The well-formedness rules defined in Figure 20 are largely straightforward. For environment formation, we say that an environment $\Gamma$ is well formed with respect to $\Delta$, written $\Delta \vdash \Gamma$, if all $y \in \Gamma$ has a well-formed type $T$ with respect to index environment $\Delta$.

A judgement of the form $\Delta_1 \vdash \theta : \Delta_2$, used for substitution, is derived through the application of two rules: $\text{WF-EMPTY}$ and $\text{WF-VAR}$. The remaining rules check the well-formedness of components from index types and propositions.

\begin{align*}
\Delta & \vdash \Gamma \quad \text{(WF-EMPTY)} \quad \Delta \vdash T \quad \Delta_1, y \notin \Gamma, \Delta \vdash T, y : T \quad \text{(WF-\Gamma)} \\
\Delta_1 & \vdash \theta : \Delta_2 \quad \Delta_1 \vdash \theta : \Delta_2, x \notin \theta \quad \Delta_1 \vdash i : I \quad \text{(WF-VAR)}
\end{align*}

\begin{align*}
\Delta & \vdash () \quad \text{(WF-EMPTY)} \quad \Delta_1 \vdash () \quad \Delta \vdash () \quad \text{(WF-\theta-Empty)} \\
\Delta & \vdash I \quad \text{(WF-INT)} \quad \Delta \vdash \text{integer} \quad \Delta_1 \vdash I \quad \Delta_2 \vdash \{x : I \mid p\} \quad \text{(WF-SUBSET)}
\end{align*}

\begin{align*}
\Delta & \vdash p \quad \Delta_1 \vdash p_1 \quad \Delta_2 \vdash p_2 \quad \text{(WF-CONJ)} \quad \Delta \vdash \text{integer} \quad \Delta \vdash j : \text{integer} \quad \Delta \vdash i \leq j \quad \text{(WF-INEQ)}
\end{align*}

Fig. 20. Well-formedness rules

A Additional Definitions

We present many technical parts of DOL in Sections 5 and 6. In this appendix, we consider additional details that did not fit in the technical report. Then, in Appendices C and D, we prove the main result of our language, type soundness via progress and subject reduction.

A.1 Well-Formedness

The rules in Figure 21 are mostly adapted from Xi [26,31], except for $\text{I-SUB}$, since DML does not include subtyping for index types. $\text{I-SUB}$ is the standard subsumption rule which allow us to introduce in DOL subtyping for index types, given by the rules in Figure 22. In rule $\text{I-Intro}$, the semantic judgement $\Delta \models p[i/x]$
To prove subtyping equivalence (Appendix B) and the main result of the paper, type soundness via subject reduction properties and progress (Appendices C and D), we start by establishing a number of auxiliary results, including permutation, weakening, substitution, heap update and agreement between the type environment and the heap. We also define a replacement lemma and subterm typing lemma, similar to lemmas used by Wright and Felleisen [25].

We write $\mathcal{J}$ to denote an arbitrary judgement, i.e.

$$\mathcal{J} ::= I \mid T \mid I <: J \mid T <: U \mid i : I \mid t : T \Downarrow \Gamma$$
We also write $\text{FV}(\mathcal{J})$ to denote all index variables that occur in a non-bound position within $\mathcal{J}$.

**Property 1** (Permutation in $\models$ relations). If $\Delta_1 \vdash I$ and $\Delta_1, \Delta_2, x : I \models p$, then $\Delta_1, x : I, \Delta_2 \models p[i/x]$.

**Property 2** (Weakening in $\models$ relations). If $\Delta \vdash I$ and $x \notin \Delta$ and $\Delta \models p$, then $\Delta, x : I \models p[i/x]$.

**Property 3** (Substitution in $\models$ relations). If $\Delta \vdash i : I$ and $\Delta, x : I \models p$, then $\Delta \models p[i/x]$.

**Property 4** (Reflexivity in $\models$ relations). If $\Delta \vdash i : I$ and $\Delta \vdash j : I$, then $\Delta \models i \leq j$ and $\Delta \models j \leq i$.

**Lemma 1** ($\Delta$-Permutation). Suppose $\Delta_1 \vdash I$.

1. If $\Delta_1, \Delta_2, x : I \vdash \mathcal{J}$, then $\Delta_1, x : I, \Delta_2 \vdash \mathcal{J}$.
2. If $\Delta_1, \Delta_2, x : I; \Gamma_1 \vdash \mathcal{J}$, then $\Delta_1, x : I, \Delta_2; \Gamma_1 \vdash \mathcal{J}$.

**Proof.** By mutual induction on the derivation of the judgements. We show case $\text{WF-INEQ}$; the remaining cases are similar.

\[
\mathcal{D} = \frac{\Delta_1, \Delta_2, x : I \vdash i : \text{integer}}{\Delta_1, \Delta_2, x : I \vdash i \leq j} \quad \text{(WF-INEQ)}
\]

$\Delta_1, x : I, \Delta_2 \vdash i : \text{integer}$  
By the IH

$\Delta_1, x : I, \Delta_2 \vdash j : \text{integer}$  
By the IH

$\Delta_1, x : I, \Delta_2 \vdash i \leq j$  
By $\text{WF-INEQ}$

**Lemma 2** ($\Gamma$-Permutation). If $\Delta; \Gamma_1, \Gamma_2, y : T_1 \vdash t : T \vdash \Gamma_3, \Gamma_4, y : T_2$, then $\Delta; \Gamma_1, \Gamma_2, y : T_1, \Gamma_2 \vdash t : T \vdash \Gamma_3, \Gamma_4, y : T_2, \Gamma_4$.

**Proof.** By induction on the derivation of the judgement $\Delta; \Gamma_1, \Gamma_2, y : T_1 \vdash t : T \vdash \Gamma_3, \Gamma_4, y : T_2$. We show case $\text{T-FIELD}$; the remaining cases are similar.

\[
\mathcal{D} = \frac{\Delta; \Gamma_1, \Gamma_2, y : T_1 \vdash y' : U \vdash \Gamma_3, \Gamma_4, y : T_2}{\Delta; \Gamma_1, \Gamma_2, y : T_1 \vdash y'.l : T \vdash \Gamma_3, \Gamma_4, y : T_2} \quad \text{(T-FIELD)}
\]

$\Gamma_1 = \Gamma_3$ and $\Gamma_2 = \Gamma_4$  
From $\mathcal{D}$

$T = U.l$  
From $\mathcal{D}$

$\Delta; \Gamma_1, y : T_1, \Gamma_2 \vdash y' : U \vdash \Gamma_3, \Gamma_4, y : T_2 \Gamma_4$  
By the IH

$\Delta; \Gamma_1, y : T_1, \Gamma_2 \vdash y'.l : T \vdash \Gamma_3, \Gamma_4, y : T_2 \Gamma_4$  
By $\text{T-FIELD}$
Lemma 3 (Free Variable Containment). If $\Delta \vdash J$, then $\text{FV}(J) \subseteq \Delta$.

Proof. By mutual induction on the derivation of the judgement $\Delta \vdash J$. The proof is similar to the one of Lemma 1.

Lemma 4 ($\Delta$-Weakening). Suppose $\Delta \vdash I$ and $x \notin \Delta$.

1. If $\Delta \vdash J$, then $\Delta, x : I \vdash J$.
2. If $\Delta; \Gamma \vdash J$, then $\Delta, x : I; \Gamma \vdash J$.

Proof. By mutual induction on the derivation of the judgements $\Delta \vdash J$ and $\Delta; \Gamma \vdash J$. We present the most representative cases; the omitted ones are similar.

Well-Formedness. All the cases for well-formedness of ordinary types follow directly by induction. We consider below the two cases for index type formation.

Case WF-INTEGER. Immediate: $\Delta, x : I \vdash \text{integer}$.

Case WF-SUBSET.

$$D = \frac{\Delta \vdash J \quad \Delta, x' : J \vdash p}{\Delta \vdash \{x' : J \mid p\}} \quad \text{(WF-SUBSET)}$$

$$\Delta, x : I \vdash J \quad \text{By the IH}$$
$$\Delta, x' : J, x : I \vdash p \quad \text{By the IH}$$
$$x \notin \text{FV}(J) \quad \text{From Lemma 3, FV}(I) \subseteq \Delta$$
$$\Delta, x : I, x' : J \vdash p \quad \text{By Property 2}$$
$$\Delta, x : I \vdash \{x' : J \mid p\} \quad \text{By WF-SUBSET}$$

Subtyping. The cases for subtyping of ordinary types follow directly by induction. We consider the three cases for index subtyping.

Case S-INTEGER. The result $\Delta, x : I \vdash \text{integer} <: \text{integer}$ is immediate.

Case S-SUBSETL.

$$D = \frac{\Delta \vdash J_1 <: J_2 \quad \Delta \vdash \{x' : J_1 \mid p\}}{\Delta \vdash J_1 <: \{x' : J_2 \mid p\}} \quad \text{(S-SUBSETL)}$$

$$\Delta, x : I \vdash J_1 <: J_2 \quad \text{By the IH}$$
$$\Delta, x : I \vdash \{x' : J_1 \mid p\} \quad \text{By the IH}$$
$$\Delta, x : I \vdash \{x' : J_1 \mid p\} <: J_2 \quad \text{By S-SUBSETL}$$

Case S-SUBSETR.

$$D = \frac{\Delta \vdash J_1 <: J_2 \quad \Delta, x' : J_2 \vdash p}{\Delta \vdash J_1 <: \{x' : J_2 \mid p\}} \quad \text{(S-SUBSETR)}$$

$$\Delta, x : I \vdash J_1 <: J_2 \quad \text{By the IH}$$
$$\Delta, x' : J_2, x : I \vdash p \quad \text{By Property 2}$$
$$x \notin \text{FV}(J_2) \quad \text{From Lemma 3, FV}(I) \subseteq \Delta$$
$$\Delta, x : I, x' : J_2 \vdash p \quad \text{By Property 1}$$
$$\Delta, x : I \vdash J_1 <: \{x' : J_2 \mid p\} \quad \text{By S-SUBSETR}$$
Index Terms. There are five cases to consider. We present the two most representative ones. The cases of rules I-Sum and I-Max follow directly by induction.

Case I-INTEGER. Immediate: $\Delta, x : I \vdash n : \text{integer}$.

Case I-VAR.

\[ D = \frac{x' : J \in \Delta}{\Delta \vdash x' : J} \quad (\text{I-VAR}) \]

\[ x' : J \in (\Delta, x : I) \quad \text{By definition} \]

\[ \Delta, x : I \vdash x' : J \quad \text{By I-VAR} \]

Case I-Intro.

\[ D = \frac{\Delta \vdash i : J \quad \Delta, x' : J \vdash p \quad \Delta \vdash p[i/x]}{\Delta \vdash i : \{x' : J | p\}} \quad (\text{I-Intro}) \]

\[ \Delta, x : I \vdash i : J \quad \text{By the IH} \]

\[ \Delta, x' : J, x : I, \vdash p \quad \text{By the IH} \]

\[ x \notin \text{FV}(J) \quad \text{From Lemma 3} \]

\[ \Delta, x : I, x' : J \vdash p \quad \text{By permutation (Lemma 1)} \]

\[ \Delta, x : I \models p[i/x] \quad \text{By Property 2} \]

\[ \Delta, x : I \vdash i : \{x' : J | p\} \quad \text{By I-Intro} \]

Ordinary Terms. All the cases follow directly by induction.

Lemma 5 (I-Weakening). If $\Delta \vdash T'$ and $y' \notin \Gamma_1$ and $\Delta; \Gamma_1 \vdash t : T \vdash \Gamma_2$, then $\Delta; \Gamma_1, y' : T' \vdash t : T \vdash \Gamma_2, y' : T'$.

Proof. By induction on the derivation of the judgement $\Delta; \Gamma_1 \vdash t : T \vdash \Gamma_2$. We show some cases; the omitted ones are similar.

Case T-VAR.

\[ D = \frac{\Delta \vdash \Gamma_2 \quad y : T \in \Gamma_2}{\Delta; \Gamma_1, y : T \vdash \Gamma_2} \quad (\text{T-VAR}) \]

\[ \Gamma_2 = \Gamma_1 \quad \text{From } D \]

\[ \Delta \vdash \Gamma_2, y' : T' \quad \text{By Wf-Gamma} \]

\[ y : T \in (\Gamma_2, y' : T') \quad \text{By definition} \]

\[ \Delta; \Gamma_1, y' : T' \vdash y : T \vdash \Gamma_2, y' : T' \quad \text{By T-VAR} \]

Case T-FIELD.

\[ D = \frac{\Delta; \Gamma_1 \vdash y : U \vdash \Gamma_2}{\Delta; \Gamma_1 \vdash y, l : T \vdash \Gamma_2} \quad (\text{T-FIELD}) \]
\[ \Gamma_2 = \Gamma_1 \quad \text{From } D \]
\[ T = U.l \quad \text{From } D \]
\[ \Delta; \Gamma_1, y': T' \vdash y : U \vdash \Gamma_2, y': T' \quad \text{By the IH} \]
\[ \Delta; \Gamma_1, y': T' \vdash y.l : T \vdash \Gamma_2, y': T' \quad \text{By } T\text{-FIELD} \]

**Case T-UnAssign.**
\[ D = \frac{\Delta; \Gamma_1 \vdash s : C \vdash \Gamma_1}{\Delta; \Gamma_1 \vdash y.l : C \vdash \Gamma_1} \quad (T\text{-UnAssign}) \]
\[ \Gamma_2 = \Gamma_1 \quad \text{From } D \]
\[ T = \text{Empty} \quad \text{From } D \]
\[ \Delta; \Gamma_1, y': T' \vdash s : C \vdash \Gamma_1, y': T' \quad \text{By the IH} \]
\[ \Delta; \Gamma_1, y': T' \vdash y.l : C \vdash \Gamma_1, y': T' \quad \text{By the IH} \]
\[ \Delta; \Gamma_1, y': T' \vdash y.l := s : T \vdash \Gamma_2, y': T' \quad \text{By } T\text{-UnAssign} \]

**Case T-LinAssign.**
\[ D = \frac{\Delta; \Gamma_1 \vdash s : U \vdash \Gamma_1 \{s \mapsto \text{Empty}\}}{\Delta; \Gamma_1 \vdash y.l : U \vdash \Gamma_1 \{s \mapsto \text{Empty}\}} \quad (T\text{-LinAssign}) \]
\[ \Gamma_2 = \Gamma_1 \{y.l \mapsto U\}' \{s \mapsto \text{Empty}\} \quad \text{From } D \]
\[ T = \text{Empty} \quad \text{From } D \]
\[ \Delta; \Gamma_1, y': T' \vdash s : U' \vdash \Gamma_1, y': T' \quad \text{By the IH} \]
\[ \Delta; \Gamma_1, y': T' \vdash y.l : U \vdash \Gamma_1, y': T' \quad \text{By the IH} \]
\[ \Delta; \Gamma_1, y': T' \vdash y.l := s : T \vdash \Gamma_2, y': T' \quad \text{By } T\text{-LinAssign} \]

**Case T-NewAssign.**
\[ D = \frac{\Delta; \Gamma_1 \vdash \bar{s} \mapsto \bar{U}' \theta \vdash \Gamma_1 \quad \Delta; \Gamma_1 \vdash y.l : T_1 \vdash \Gamma_1}{\Delta; \Gamma_1 \vdash y.l := (\text{new } C)\bar{s} : T \vdash \Gamma_2} \quad (T\text{-NewAssign}) \]
\[ C, \text{init} = \Pi \Delta'; T_2 \times \bar{U} \sim \bar{U}' \quad \Delta \vdash \theta : \Delta' \]
\[ \Gamma_2 = \Gamma_1 \{\bar{s} \mapsto \bar{U}' \theta \} \{y.l \mapsto T_2 \theta\} \quad \text{From } D \]
\[ T = \text{Empty} \quad \text{From } D \]
\[ \Delta; \Gamma_1, y': T' \vdash \bar{s} : \bar{U} \theta \vdash \Gamma_1, y': T' \quad \text{By the IH} \]
\[ \Delta; \Gamma_1, y': T' \vdash y.l : T_1 \vdash \Gamma_1, y': T' \quad \text{By the IH} \]
\[ \Delta; \Gamma_1, y': T' \vdash y.l := (\text{new } C)\bar{s} : T \vdash \Gamma_2, y': T' \quad \text{By } T\text{-NewAssign} \]

**Case T-Call.**
\[ D = \frac{\Delta \vdash \theta : \Delta' \quad \Delta \vdash T'' < T_1 \theta \quad \Delta; \Gamma_1 \vdash s_1 : U_1 \theta \vdash \Gamma_1}{\Delta; \Gamma \vdash (s_1.l)s_2 : T \vdash \Gamma_2} \quad (T\text{-Call}) \]
\[ \Delta; \Gamma_1 \vdash s_1 : T'' \vdash \Gamma_1 \quad T''.l = \Pi \Delta'.(T_1 \sim T_2) \times (U_1 \sim U_2) \]
Lemma 6 (Substitution of Index Terms). Suppose $\Delta \vdash i : I$.

1. If $\Delta, x : I \vdash J$, then $\Delta \vdash J[i/x]$.
2. If $\Delta, x : I; \Gamma \vdash J$, then $\Delta; \Gamma[i/x] \vdash J[i/x]$.

Proof. By mutual induction on the derivation of the judgements. We present some cases for algorithmic subtyping; the remaining cases are similar.

Case SA-$\Sigma$L.

\[
\frac{\Delta, x : I \vdash x' : J \vdash T <_{\A} U \quad \Delta, x : I \vdash \Sigma x' : J,T <_{\A} U}{\Delta, x : I \vdash \langle \Sigma x' : J,T \rangle[i/x] <_{\A} U[i/x]} \tag{SA-$\Sigma$L}
\]

By the IH

\[
\frac{\Delta \vdash x' : J \vdash T[i/x] <_{\A} U[i/x]}{\Delta \vdash \Sigma x' : J,T[i/x] <_{\A} U[i/x]} \tag{SA-$\Sigma$L}
\]

By definition

Case SA-$\Sigma$R.

\[
\frac{\Delta, x : I \vdash \langle \Sigma x' : J,T \rangle[i/x] <_{\A} U[i/x]}{\Delta, x : I \vdash \langle \Sigma x' : J \vdash T[i/x] \rangle[i/x] <_{\A} U[i/x]} \tag{SA-$\Sigma$R}
\]

By definition

\[
\frac{\Delta \vdash x' : J \vdash T[i/x]}{\Delta \vdash \langle \Sigma x' : J,T \rangle[i/x] <_{\A} U[i/x]} \tag{SA-$\Sigma$R}
\]

By definition

Lemma 7 (Substitution of Program Terms). If $\Delta; \Gamma_1, y : T_1 \vdash t : T \vdash \Gamma_2, y : T_2$ and $\Delta; \Gamma \vdash o : T_1 \vdash \Gamma$ and $o \notin \Gamma_1$, then $\Delta; \Gamma_1, o : T_1 \vdash t[\gamma/y] : T \vdash \Gamma_2, o : T_2$.

Proof. By induction on the derivation of judgement $\Delta; \Gamma_1, y : T_1 \vdash t : T \vdash \Gamma_2, y : T_2$. We show some cases; the omitted ones are similar.

Case T-VAR. If $t = y'$, there are two subcases depending on whether $y'$ is $y$ or some other variable. If $y' = y$, then $y'[\gamma/y] = o$. The required result is $\Delta; \Gamma_1, o : T_1 \vdash o : T_1 \vdash \Gamma_2, o : T_2$, with $\Gamma_2 = \Gamma_1$ and $T_2 = T_1$. The other case is when $y' \neq y$, then $y'[\gamma/y] = y'$, and the result is immediate.
Case T-FIELD.

\[
\Delta; \Gamma_1, y : T_1 \vdash y' : T' \vdash \Gamma_2, y : T_2 \\
\Delta; \Gamma_1, y : T_1 \vdash y.l : T \vdash \Gamma_2, y : T_2
\]  (T-FIELD)

\[T = T', l \text{ and } T_2 = T_1\]  \quad \text{From } \mathcal{D}

\[\Gamma_2 = \Gamma_1\]  \quad \text{From } \mathcal{D}

\[\Delta; \Gamma_1, o : T_1 \vdash y'[^o/y] : T' \vdash \Gamma_2, o : T_2\]  \quad \text{By the IH}

\[\Delta; \Gamma_1, o : T_1 \vdash (y'.l[^o/y]) : T \vdash \Gamma_2, o : T_2\]  \quad \text{By T-FIELD}

Case T-UNASSIGN.

\[
\Delta; \Gamma_1, y : T_1 \vdash s : C \vdash \Gamma_2, y : T_2 \\
\Delta; \Gamma_1, y : T_1 \vdash y'.l : C \vdash \Gamma_2, y : T_2 \\
\Delta; \Gamma_1, y : T_1 \vdash y.l \:= s : T \vdash \Gamma_2, y : T_2
\]  (T-UNASSIGN)

\[T = \text{Empty and } T_2 = T_1\]  \quad \text{From } \mathcal{D}

\[\Gamma_2 = \Gamma_1\]  \quad \text{From } \mathcal{D}

\[\Delta; \Gamma_1, o : T_1 \vdash s[^o/y] : C \vdash \Gamma_2, o : T_2\]  \quad \text{By the IH}

\[\Delta; \Gamma_1, o : T_1 \vdash y'.l[^o/y] : C \vdash \Gamma_2, o : T_2\]  \quad \text{By the IH}

\[\Delta; \Gamma_1, o : T_1 \vdash y.l := s[^o/y] : T \vdash \Gamma_2, o : T_2\]  \quad \text{By T-UNASSIGN}

\[\Delta; \Gamma_1, o : T_1 \vdash (y'.l := s)[^o/y] : T \vdash \Gamma_2, o : T_2\]  \quad \text{By definition}

Case T-LINASSIGN.  This case is similar to the one above, except for the types of the premises and the shape of the concluding environment: \(\Gamma_2 = \Gamma_1\{y'.l \mapsto T\}\{s \mapsto \text{Empty}\} \).

Case T-NEWASSIGN.

\[
C, \text{init} = \Pi \Delta'. T_1 \times \bar{U} \leadsto \bar{U}' \quad \Delta \vdash \theta : \Delta'
\]

\[
\mathcal{D} = \frac{\Delta; \Gamma_1, y : T_1 \vdash s.l : \bar{U}, \bar{U} : \theta, T_3 \vdash \Gamma_1, y : T_2}{\Delta; \Gamma_1, y : T_1 \vdash y.l := (\text{new } C)s.l \vdash T \vdash \Gamma_2}
\]  (T-NEWASSIGN)

\[T = \text{Empty}\]  \quad \text{From } \mathcal{D}

\[\Gamma_2 = \Gamma_1\{s \mapsto \bar{U}'\}\{y.l \mapsto T_3 \theta\}\]  \quad \text{From } \mathcal{D}

\[\Delta; \Gamma_1, o : T_1 \vdash s[^o/y] : \bar{U} \vdash \Gamma_1, o : T_2\]  \quad \text{By the IH}

\[\Delta; \Gamma_1 \vdash y'.l[^o/y] : T_3 \vdash \Gamma_1\]  \quad \text{By the IH}

\[\Delta; \Gamma_1, o : T_1 \vdash (y'.l := (\text{new } C)s)[^o/y] : T \vdash \Gamma_2, o : T_2\]  \quad \text{By T-NEWASSIGN}

\[\Delta; \Gamma_1, o : T_1 \vdash (y.l := (\text{new } C)s)[^o/y] : T \vdash \Gamma_2, o : T_2\]  \quad \text{By definition}

Case T-CALL.

\[\Delta; \Gamma_1, y : T_1 \vdash s_1 : U \vdash \Gamma_1, y : T_2 \quad U.l = \Pi \Delta' .(T' \leadsto T'') \times (U_1 \leadsto U_2)\]

\[\Delta \vdash \theta : \Delta' \quad \Delta' \vdash U < T' \theta \quad \Delta; \Gamma_1, y : T_1 \vdash s_2 : U_1 \theta \vdash \Gamma_1, y : T_2\]

\[
\Delta; \Gamma_1, y : T_1 \vdash (s_1.l)s_2 : T \vdash \Gamma_2, y : T_2
\]  (T-CALL)

36
Lemma 8 (Heap Agreement). If $\Delta; \Gamma \vdash h$ and $o : T \in \Gamma$, then $h(o) = R$ and $\Delta; \Gamma \vdash R.l_k : T.l_k \vdash \Gamma$ for all $1 \leq k \leq m$.

Proof. The proof follows from the shape of rule T-HEAPOBJECT.

Using permutation (Lemma 2) on $\Delta; \Gamma \vdash h$, we can build a derivation whose conclusion introduces $o$ by T-HEAPOBJECT. From the shape of the conclusion of the rule, we have $\Gamma = \Gamma', o : T$ and $h = (h', o = R)$ as follows:

$$
\frac{
\Delta; \Gamma' \vdash h'
}{\Delta; \Gamma', o : T \vdash h', o = R}
$$

(T-HEAPOBJECT)

Since $\Gamma(o) = T$, we can also build a derivation whose conclusion is $\Delta \vdash \Gamma', o : T$.

$$
\frac{
\Delta \vdash \Gamma'
\quad o \notin \Gamma'
}{\Delta \vdash \Gamma', o : T}
$$

(WF-Γ)

By the premises of HEAPOBJECT and WF-Γ and weakening (Lemma 5), we conclude $\Delta; \Gamma \vdash R.l_k : T.l_k \vdash \Gamma$ for all $1 \leq k \leq m$, as required.

Lemma 9 (Heap Operations Preserve Variable Types). If $\Delta; \Gamma \vdash s : T \vdash \Gamma$ and $\Gamma' = \Gamma(s' \rightarrow T')$, then $\Delta; \Gamma' \vdash s : T \vdash \Gamma'$.

Proof. By case analysis on the form of $s$.

Lemma 10 (Heap Update). If $\Delta; \Gamma \vdash h$ and $\Delta; \Gamma \vdash o' : T' \vdash \Gamma$ and $\Delta; \Gamma \vdash o.l : T \vdash \Gamma$, then $\Delta; \Gamma \vdash o.l \rightarrow T' \vdash h(o.l \rightarrow o')$.

Proof. The proof follows from the definitions of heap and environment update. From $\Delta; \Gamma \vdash o.l : T \vdash \Gamma$, we may build the following derivation tree:

$$
\frac{
\Delta \vdash \Gamma
\quad o \in \Gamma
}{\Delta; \Gamma \vdash o : T \vdash \Gamma}
$$

(T-VAR)

$$
\frac{
\Delta \vdash \Gamma
\quad T \in \Gamma
}{\Delta; \Gamma \vdash T.l \vdash \Gamma}
$$

(T-FIELD)

$$
\frac{
\Delta \vdash \Gamma
\quad T \in \Gamma
}{\Delta; \Gamma \vdash T \vdash \Gamma}
$$

(T-SUB)

From $\Delta \vdash \Gamma$ and $o \in \Gamma$, we have $\Delta \vdash T_1$. Let $\Gamma_1 = \Gamma\{o.l \rightarrow T'\}$ and $h_1 = h\{o.l \rightarrow o'\}$. We can use permutation (Lemma 2) on $\Delta; \Gamma_1 \vdash h_1$ to build a derivation whose conclusion introduces $o$ by T-HEAPOBJECT as follows:
\[ \Delta; \Gamma' \vdash h' \quad \forall 1 \leq k \leq m \quad \Delta; \Gamma' \vdash R.l_k : T_1.l_k \vdash \Gamma'' \]  
\[ \frac{}{\Delta; \Gamma', o : T_1 \{ l \mapsto T' \} \vdash h', o = R \{ l \mapsto o \}} \]  
(T-\text{HEAPOBJECT})

To prove \( \Delta; \Gamma_1 \vdash o' : T' \vdash \Gamma_1 \), we need to consider two cases. First, if \( o' \neq R.l \), then by definition of \( \Gamma_1 \), we have \( \Gamma_1(o') = \Gamma'(o') \). The result can be proved using \( \Delta \vdash T_1 \) together with \( \Delta; \Gamma' \vdash o' : T' \vdash \Gamma'' \) and weakening (Lemma 5), followed by an application of Lemma 9. On the other hand, if \( o' = R.l \), then it suffices to show that \( \Delta; \Gamma_1 \vdash R.l_k : T_1.l_k \vdash \Gamma_1 \) holds for all \( 1 \leq k \leq m \). This may be proved using weakening (Lemma 5) on the premises of the derivation of T-\text{HEAPOBJECT} to obtain \( \Delta; \Gamma \vdash R.l_k : T_1.l_k \vdash \Gamma \), followed by an application of Lemma 9, noting that by definition of \( \Gamma_1 \), we have \( \Gamma'((o.l) = T' \).

**Lemma 11 (Subterm Typing in \( \mathcal{E} \)).** If \( D_1 \) is a derivation of the judgement \( \Delta; \Gamma_1 \vdash \mathcal{E}[t] : T \vdash \Gamma_2 \), then there exist \( T' \) and \( \Gamma_3 \) such that \( D_1 \) has a subderivation \( D_2 \) concluding \( \Delta; \Gamma_1 \vdash t : T' \vdash \Gamma_3 \), and the position of \( D_2 \) in \( D_1 \) corresponds to the position of the hole in \( \mathcal{E} \).

**Proof.** By induction on the structure of \( \mathcal{E}[t] \).

**Lemma 12 (Replacement in \( \mathcal{E} \)).** If

1. \( D_1 \) is a derivation concluding \( \Delta; \Gamma_1 \vdash \mathcal{E}[t_1] : T_1 \vdash \Gamma_2 \)
2. \( D_2 \) is a subderivation of \( D_1 \) concluding \( \Delta; \Gamma_3 \vdash t_1 : T_2 \vdash \Gamma_4 \)
3. the position of \( D_2 \) in \( D_1 \) corresponds to the position of the hole in \( \mathcal{E} \)
4. \( \Delta; \Gamma_3 \vdash t_2 : T_2 \vdash \Gamma_4 \)

then \( \Delta; \Gamma_3 \vdash \mathcal{E}[t_2] : T_1 \vdash \Gamma_2 \).

**Proof.** Let \( D_3 \) be the derivation concluding \( \Delta; \Gamma_3 \vdash t_2 : T_2 \vdash \Gamma_4 \). Replace the subtree \( D_2 \) with \( D_3 \) in \( D_1 \), as well as all the relevant occurrences of \( t_1 \) with \( t_2 \). The resulting tree is a valid derivation concluding \( \Delta; \Gamma_3 \vdash \mathcal{E}[t_2] : T_1 \vdash \Gamma_2 \).

**B Proof of Theorem 1, Subtyping Equivalence**

To prove subtyping equivalence, we need to define a few more auxiliary lemmas, including soundness and completeness of algorithmic subtyping, with the latter requiring two lemmas stating the reflexivity and transitivity of the algorithmic subtyping relation.

**Lemma 13 (Soundness of Algorithmic Subtyping).** If \( \Delta \vdash T <:_{A} U \), then \( \Delta \vdash T <: U \).

**Proof.** By induction on the derivation of \( \Delta \vdash T <:_{A} U \).

**Case SA-SUBCLASS.** Immediate.

**Case SA-CLASS.** \( T = C = U \). By S-CLASSL/R, we have \( \Delta \vdash C <: \text{ctype}(C) \) and \( \Delta \vdash \text{ctype}(C) <: C \). Then, by S-TRANS, we obtain \( \Delta \vdash C <: C \), i.e. \( \Delta \vdash T <: U \).
Case **SA-App.** \( T = T'i \) and \( U = U'j \). By the premises of SA-App and the induction hypothesis, we have \( \Delta \vdash T' <: U' \) and \( \Delta \vdash i \leq j \). Then, by S-App, we obtain \( \Delta \vdash T'i <: U'j \).

Case **SA-AppL.** \( T = Ci \) and \( U = \{l_k : T_k^{\ell_1 \ldots \ell_n} \} \). By the premises of SA-AppL, we have \( \Delta \vdash \text{ctype}_A(C)i \Downarrow \beta U' \) and \( \Delta \vdash U' <: A \{l_k : T_k^{\ell_1 \ldots \ell_n} \} \). By the reflexive transitive closure of \( \rightarrow^\beta \) and the induction hypothesis, we obtain \( \Delta \vdash \text{ctype}(C)\bar{i} <: \{l_k : T_k^{\ell_1 \ldots \ell_n} \} \). Then, by S-Trans, we get \( \Delta \vdash C\bar{i} <: \{l_k : T_k^{\ell_1 \ldots \ell_n} \} \).

Case **SA-AppR.** Similar to the previous case.

Case **SA-IRL.** \( T = \Pi x : I.T' \). By the premises of SA-IRL, we have \( \Delta \vdash i : I \) and by the induction hypothesis, we get \( \Delta \vdash T'[i/x] <: U' \). By S-IRL, we obtain \( \Delta \vdash \Pi x : I.T' <: U' \).

Case **SA-IRR.** \( U = \Pi x : I.U' \). By the premise of SA-IRR and the induction hypothesis, we have \( \Delta, x : I \vdash T <: U' \). By S-IRR, we obtain \( \Delta \vdash T <: \Pi x : I.U' \).

Case **SA-IRL.** Similar to SA-IRR.

Case **SA-ΣR.** Similar to SA-IRL.

Case **SA-Record.** \( T = \{l_k : T_k^{\ell_1 \ldots \ell_n+m} \} \) and \( U = \{l_k : U_k^{\ell_1 \ldots \ell_n} \} \). By the premises of SA-Record and the induction hypothesis, we have \( \forall 1 \leq k \leq n, \Delta \vdash T_k := U_k \). Then, by S-Record, we obtain \( \Delta \vdash \{l_k : T_k^{\ell_1 \ldots \ell_n} \} <: \{l_k : U_k^{\ell_1 \ldots \ell_n} \} \).

Case **SA-→.** \( T = (T_1 \rightarrow T_2) \) and \( U = (U_1 \rightarrow U_2) \). By the premises of SA-→ and the induction hypothesis, we have \( \Delta \vdash T_1 <: U_1 \) and \( \Delta \vdash T_2 <: U_2 \). Then, by S-→, we obtain \( \Delta \vdash (T_1 \rightarrow T_2) <: (U_1 \rightarrow U_2) \).

Case **SA-×.** Similar to the previous case.

**Lemma 14 (ΠL Substitution).** If \( \Delta \vdash T[i/x] <: A U \) and \( \Delta \vdash i : I \), then \( \Delta \vdash \Pi x : I.T <: A U \).

**Proof.** By structural induction on \( U \), examining all the possible cases.

**Case** \( U = U^o \). By SA-ΠL.

**Case** \( U = \Pi x' : J.U' \).

\[
\begin{align*}
\Delta \vdash T[i/x] <: A \Pi x' : J.U' & \quad \text{Given} \\
\Delta, x' : J \vdash T[i/x] <: A U' & \quad \text{By inversion} \\
\Delta, x' : J \vdash \Pi x : I.T <: A U' & \quad \text{By the IH} \\
\Delta \vdash \Pi x : I.T <: A \Pi x' : J.U' & \quad \text{By SA-ΠR}
\end{align*}
\]

**Lemma 15 (ΣR Substitution).** If \( \Delta \vdash T <: U[i/x] \) and \( \Delta \vdash i : I \), then \( \Delta \vdash T <: \Sigma x : I.U \).

**Proof.** By structural induction on \( T \), examining all the possible cases.

**Case** \( T = T^o \). By SA-ΣR.
Case \( T = \Sigma x' : J.U' \).

\[
\begin{align*}
\Delta \vdash \Sigma x' : J.U' <: U'[\ell/x] & \quad \text{Given} \\
\Delta, x' : J \vdash U' <: \mathcal{A} U'[\ell/x] & \quad \text{By inversion} \\
\Delta, x' : J \vdash U' <: \mathcal{A} \Sigma x : I.U & \quad \text{By the IH} \\
\Delta \vdash \Sigma x' : J.U' <: \mathcal{A} \Sigma x : I.U & \quad \text{By SA-\Sigma L}
\end{align*}
\]

Lemma 16 (Reflexivity of Algorithmic Subtyping). \( \Delta \vdash T <: \mathcal{A} T \)

Proof. By structural induction on \( T \), examining all the possible cases.

Case \( T = C \). By SA-\CLASS.

Case \( T = U.i \).

\[
\begin{align*}
\Delta \vdash U <: \mathcal{A} U & \quad \text{By the IH} \\
\Delta \models \bar{i} \leq \bar{i} & \quad \text{By Property 4} \\
\Delta \vdash U.i <: \mathcal{A} U.i & \quad \text{By SA-\APP}
\end{align*}
\]

Case \( T = \Pi x : I.U \).

\[
\begin{align*}
\Delta, x : I \vdash U <: \mathcal{A} U & \quad \text{By the IH and weakening (Lemma 4)} \\
\Delta, x : I \vdash U[\ell/x] <: \mathcal{A} U & \quad \text{By } U[\ell/x] = U \\
\Delta, x : I \vdash x : I & \quad \text{By assumption} \\
\Delta, x : I \vdash \Pi x : I.U <: \mathcal{A} \Pi x : I.U & \quad \text{By SA-\JL} \\
\Delta \vdash \Pi x : I.U <: \mathcal{A} \Pi x : I.U & \quad \text{By SA-\JIR}
\end{align*}
\]

Case \( T = \Sigma x : I.U \).

\[
\begin{align*}
\Delta, x : I \vdash U <: \mathcal{A} U & \quad \text{By the IH and weakening (Lemma 4)} \\
\Delta, x : I \vdash U <: \mathcal{A} U[\ell/x] & \quad \text{By } U[\ell/x] = U \\
\Delta, x : I \vdash x : I & \quad \text{By assumption} \\
\Delta, x : I \vdash U <: \mathcal{A} \Sigma x : I.U & \quad \text{By SA-\SR} \\
\Delta \vdash \Sigma x : I.U <: \mathcal{A} \Sigma x : I.U & \quad \text{By SA-\SL}
\end{align*}
\]

Case \( T = \{ l_k : U_k^{k \in \{ 1 \ldots n \}} \} \).

\[
\begin{align*}
\forall 1 \leq k \leq n, \Delta \vdash U_k <: \mathcal{A} U_k & \quad \text{By the IH} \\
\Delta \vdash \{ l_k : U_k^{k \in \{ 1 \ldots n \}} \} <: \mathcal{A} \{ l_k : U_k^{k \in \{ 1 \ldots n \}} \} & \quad \text{By SA-\RECORD}
\end{align*}
\]

Case \( T = T_1 \Rightarrow T_2 \).

\[
\begin{align*}
\Delta \vdash T_1 <: \mathcal{A} T_1 & \quad \text{By the IH} \\
\Delta \vdash T_2 <: \mathcal{A} T_2 & \quad \text{By the IH} \\
\Delta \vdash T_1 \Rightarrow T_2 <: \mathcal{A} T_1 \Rightarrow T_2 & \quad \text{By SA-\Rightarrow}
\end{align*}
\]
Case $T = T_1 \times T_2$.

$$
\Delta \vdash T_1 <_{\mathcal{A}} T_1 \quad \text{By the IH}
$$

$$
\Delta \vdash T_2 <_{\mathcal{A}} T_2 \quad \text{By the IH}
$$

$$
\Delta \vdash T_1 \times T_2 <_{\mathcal{A}} T_1 \times T_2 \quad \text{By \text{SA-\times}}
$$

Lemma 17 (Transitivity of Algorithmic Subtyping). If $\mathcal{D}_1$ is a derivation concluding $\Delta \vdash T_1 <_{\mathcal{A}} T_2$ and $\mathcal{D}_2$ is a derivation concluding $\Delta \vdash T_2 <_{\mathcal{A}} T_3$, then $\Delta \vdash T_1 <_{\mathcal{A}} T_3$.

Proof. By induction on the derivations $\mathcal{D}_1$ and $\mathcal{D}_2$.

If either derivation concludes with \text{SA-CLASS}, then the result is immediate. If $\mathcal{D}_1$ concludes with \text{SA-SUBLASS}, then $T_1 = C \bar{i}$ and $T_2 = D$. Since $\Delta \vdash D <_{\mathcal{A}} D$, it follows that $T_2 = T_3$ and $\Delta \vdash T_1 <_{\mathcal{A}} T_3$.

If $\mathcal{D}_2$ ends with \text{SA-IIR}, then $T_3 = \Pi x : I.U$. If $\mathcal{D}_2$ ends with \text{SA-\Sigma R}, then $T_3 = \Sigma x : I.U$.

We now use case analysis on the last rule used to derive $\mathcal{D}_1$, referring the cases given above when required.

**Case \text{SA-SUBLASS.}** $T_1 = C \bar{i}$ and $T_2 = D$. Three rules can conclude with a $D$ type on the left of $<_{\mathcal{A}}$.

1. Subcase \text{SA-CLASS:} Given above.
2. Subcase \text{SA-IIR:} Given above.
3. Subcase \text{SA-\Sigma R:} Given above.

**Case \text{SA-APPL.}** $T_1 = C \bar{i}$ and $T_2 = \{l_k : U_k \, k \in 1...n\}$. Four rules can conclude with a record type on the left of $<_{\mathcal{A}}$.

1. Subcase \text{SA-AppR:} By the induction hypothesis, we obtain $\Delta \vdash C \bar{i} <_{\mathcal{A}} T_3$.
2. Subcase \text{SA-IIR:} Given above.
3. Subcase \text{SA-\Sigma R:} Given above.
4. Subcase \text{SA-RECORD:} By the induction hypothesis, $\Delta \vdash \{l_k : T_k \, k \in 1...n\} <_{\mathcal{A}} T_3$.

**Case \text{SA-APP.}** $T_1 = \{l_k : U_k \, k \in 1...n\}$ and $T_2 = C \bar{i}$. Five rules can conclude with a type $C \bar{i}$ on the left of $<_{\mathcal{A}}$. 

41
1. Subcase SA-SUBCLASS: Given above.
2. Subcase SA-APP: By the induction hypothesis, \( \Delta \vdash \{ l_k : T_k \}_{k \in 1..n} \triangleleft \mathcal{A} U_j \).
3. Subcase SA-APPLE: By the induction hypothesis, \( \Delta \vdash \{ l_k : T_k \}_{k \in 1..n} \triangleleft \mathcal{A} T_3 \).
5. Subcase SA-SR: Given above.

Case **SA-IIL**. Then

\[
D_1 = \frac{\Delta \vdash T[l/x] \triangleleft \mathcal{A} U^\circ \quad \Delta \vdash i : I}{\Delta \vdash \Pi x : I.T \triangleleft \mathcal{A} U^\circ} \quad (\text{SA-IIL})
\]

\[
\begin{align*}
\Delta &\vdash U^\circ \triangleleft \mathcal{A} A_3 & \text{Given} \\
\Delta &\vdash T[l/x] \triangleleft \mathcal{A} A_3 & \text{By the IH} \\
\Delta &\vdash i : I & \text{From } D \\
\Delta &\vdash \Pi x : I.T \triangleleft \mathcal{A} A_3 & \text{By Lemma 14}
\end{align*}
\]

Case **SA-IIR**. Three rules can conclude with a \( \Pi \)-type on the left of \( \triangleleft \mathcal{A} \).

1. Subcase SA-IIR:

\[
D_1 = \frac{\Delta, x : I \vdash T_1 \triangleleft \mathcal{A} T}{\Delta \vdash T_1 \triangleleft \mathcal{A} \Pi x : I.T} \quad (\text{SA-IIR})
\]

\[
D_2 = \frac{\Delta \vdash T[l/x] \triangleleft \mathcal{A} U^\circ \quad \Delta \vdash i : I}{\Delta \vdash \Pi x : I.T \triangleleft \mathcal{A} U^\circ} \quad (\text{SA-IIL})
\]

\[
\begin{align*}
\Delta, x : I &\vdash T_1 \triangleleft \mathcal{A} T & \text{From } D_1 \\
\Delta &\vdash i : I & \text{From } D_2 \\
\Delta, x : I &\vdash T_1[l/x] \triangleleft \mathcal{A} T[l/x] & \text{By substitution (Lemma 6)} \\
\Delta &\vdash T[l/x] \triangleleft \mathcal{A} U^\circ & \text{From } D_2 \\
\Delta &\vdash T_1[l/x] \triangleleft \mathcal{A} U^\circ & \text{From the IH} \\
T_1[l/x] &\vdash T_1 & \text{From } D_1, x \notin \text{FV}(T_1) \\
\Delta &\vdash T_1 \triangleleft \mathcal{A} U^\circ & \text{From above}
\end{align*}
\]

2. Subcase SA-IIR: Given above.
3. Subcase SA-\( \Sigma \)L: Given above.

Case **SA-\( \Sigma \)L**. Similar to SA-IIR.

Case **SA-\( \Sigma \)R**. Similar to SA-IIL.

Case **SA-RECORD**. \( T_1 = \{ l_k : T_k \}_{k \in 1..n+m+p} \) and \( T_2 = \{ l_k : U_k \}_{k \in 1..n+m} \).

Two rules conclude with a record type on the left of \( \triangleleft \mathcal{A} \).

1. Subcase SA-APP: Given above.
2. Subcase SA-RECORD:

\[
D_1 = \frac{\forall 1 \leq k \leq n + m \quad \Delta \vdash T_k \triangleleft \mathcal{A} U_k}{\Delta \vdash \{ l_k : T_k \}_{k \in 1..n+m+p} \triangleleft \mathcal{A} \{ l_k : U_k \}_{k \in 1..n+m}} \quad (\text{SA-RECORD})
\]

\[
D_2 = \frac{\forall 1 \leq k \leq n \quad \Delta \vdash U_k \triangleleft \mathcal{A} U_k'}{\Delta \vdash \{ l_k : U_k \}_{k \in 1..n+m} \triangleleft \mathcal{A} \{ l_k : U_k' \}_{k \in 1..m}} \quad (\text{SA-RECORD})
\]

42
\[ \forall 1 \leq k \leq n, \Delta \vdash T_k <_{\mathcal{A}} U_k' \quad \text{By the IH} \]
\[ \Delta \vdash \{k: T_k^{k \in 1 \ldots n + m + p}\} <_{\mathcal{A}} \{k: U_k'^{k \in 1 \ldots n}\} \quad \text{By SA-RECORD} \]

Case SA-\(\rightarrow\): \(T_1 = T'_1 \rightsquigarrow T'_2\) and \(T_2 = U_1 \rightsquigarrow U_2\). Three rules can conclude with an input-output type on the left of \(<_{\mathcal{A}}\).

1. Subcase SA-\(\Pi\): Given above.
2. Subcase SA-SR: Given above.
3. Subcase SA-\(\rightleftharpoons\):

\[
D_1 = \frac{\Delta \vdash T'_1 <_{\mathcal{A}} U'_1 \quad \Delta \vdash T'_2 <_{\mathcal{A}} U'_2}{\Delta \vdash T'_1 \rightsquigarrow T'_2 <_{\mathcal{A}} U_1 \rightsquigarrow U_2} \quad \text{(SA-\(\rightleftharpoons\))} \\
D_2 = \frac{\Delta \vdash U_1 <_{\mathcal{A}} U'_1 \quad \Delta \vdash U_2 <_{\mathcal{A}} U'_2}{\Delta \vdash U_1 \rightsquigarrow U_2 <_{\mathcal{A}} U'_1 \times U'_2} \quad \text{(SA-x)}
\]

\[
\Delta \vdash T'_1 <_{\mathcal{A}} U'_1 \quad \text{By the IH} \\
\Delta \vdash T'_2 <_{\mathcal{A}} U'_2 \quad \text{By the IH} \\
\Delta \vdash T'_1 \rightsquigarrow U'_1 <_{\mathcal{A}} T'_2 \rightsquigarrow U'_2 \quad \text{By SA-x}
\]

Case SA-x: Similar to the previous case.

Lemma 18 (Completeness of Algorithmic Subtyping). If \(\Delta \vdash T <_{\mathcal{A}} U\), then \(\Delta \vdash T <_{\mathcal{A}} U\).

Proof. By induction on the derivation of \(\Delta \vdash T <_{\mathcal{A}} U\), considering the last rule.

Case S-SUBCLASS. Immediate.
Case S-CLASSL. \(T = C\) and \(U = \text{ctype}(C)\). By Lemma 17.
Case S-CLASSR. \(T = \text{ctype}(C)\) and \(U = C\). By Lemma 17.
Case S-APP. Symmetric to the corresponding case in Lemma 13.
Case S-\(\beta\). \(T = (\Pi x : I.T')i\) and \(U = T'[i/x]\). By Lemma 17.
Case S-\(\Pi\). \(T = \Pi x : I.T'\). By Lemma 16, we have \(\Delta \vdash T'[i/x] <_{\mathcal{A}} T'[i/x]\).

From the premises of S-\(\Pi\), we also have \(\Delta \vdash i : I\). Then, by Lemma 14, we obtain \(\Delta \vdash \Pi x : I.T' <_{\mathcal{A}} U\).

Case S-\(\Pi\)-\(\Pi\). \(U = \Pi x : I.U''\). From the premise of the rule and the induction hypothesis, we have \(\Delta, x : I \vdash T <_{\mathcal{A}} U\). Then, by S-\(\Pi\), we obtain \(\Delta \vdash T <_{\mathcal{A}} \Pi x : I.U''\).

Case S-\(\Sigma\)L. Similar to S-\(\Pi\).
Case S-\(\Sigma\)R. Similar to S-\(\Pi\).
Case S-REFL. \(U = T\). By Lemma 16, we have \(\Delta \vdash T <_{\mathcal{A}} U\).
Case S-TRANS. \(T = T_1\) and \(U = T_3\). By the induction hypothesis, we have \(\Delta \vdash T_1 <_{\mathcal{A}} T_2\) and \(\Delta \vdash T_2 <_{\mathcal{A}} T_3\). By Lemma 17, we obtain \(\Delta \vdash T_1 <_{\mathcal{A}} T_3\).

Theorem 1 (Subtyping Equivalence). \(\Delta \vdash T <_{\mathcal{A}} U\) iff \(\Delta \vdash T <_{\mathcal{A}} U\).

Proof. By Lemma 13 and Lemma 18.

Theorem 2 (Decidability of Subtyping). \(\Delta \vdash T <_{\mathcal{A}} U\) is decidable. Moreover, by the previous result, this extends for deciding \(\Delta \vdash T <_{\mathcal{A}} U\).

Proof. For any rule in Figure 16, each premise is smaller than the conclusion.
C Proof of Theorem 3, Subject Reduction

Theorem 3 (Subject Reduction). Let $P$ be a well-formed program ($\vdash P$). If, in this context, $\Delta; \Gamma_1 \vdash S_1 : T \vdash \Gamma_2$ and $S_1 \rightarrow S_2$, then there exists $\Gamma_3$ such that $\Delta; \Gamma_3 \vdash S_2 : T \vdash \Gamma_2$.

Proof. By rule induction on the reduction judgement $S_1 \rightarrow S_2$. We proceed by case analysis. Let $S_1 = (h, t)$. The hypothesis $\Delta; \Gamma_1 \vdash (h, t) : T \vdash \Gamma_2$ is the conclusion of T-State, whose premises are (a) $\Delta; \Gamma_1 \vdash h$ and (b) $\Delta; \Gamma_1 \vdash t : T \vdash \Gamma_2$.

**R-Context.** Then $t = E[t_1]$, and from the premise of the rule, we know

$$(h_1, t_1) \rightarrow (h_2, t_2)$$

From (b) and subterm typing (Lemma 11), we get

$$\Delta; \Gamma_1 \vdash t_1 : T' \vdash \Gamma_4$$

From (a), (2) and T-State, we derive

$$\Delta; \Gamma_1 \vdash (h_1, t_1) : T' \vdash \Gamma_4$$

From (3), (1) and the induction hypothesis, we obtain

$$\Delta; \Gamma_3 \vdash (h', t_2) : T' \vdash \Gamma_4$$

From the premises of T-State and (4), we know

$$\Delta; \Gamma_3 \vdash h'$$

$$\Delta; \Gamma_3 \vdash t_2 : T' \vdash \Gamma_4$$

Using (b), (2), (5) and replacement (Lemma 12), we get

$$\Delta; \Gamma_3 \vdash E[t_2] : T \vdash \Gamma_2$$

From (5), (7) and T-State, we conclude

$$\Delta; \Gamma_3 \vdash (h', E[t_2]) : T \vdash \Gamma_2$$

**R-Field.** Then $t = o.l$. From the premise and shape of the conclusion of (b), we know

$$\Delta; \Gamma_1 \vdash o : T_1 \vdash \Gamma_2$$

$$T = T_1.l$$

$$\Gamma_2 = \Gamma_1$$

Since (8) is derived by T-Var, we also know $o \in \Gamma_1$. Using this with (a) and agreement (Lemma 8), we obtain

$$\Gamma_1(o) = T_1$$

$$h(o) = R$$

$$\Delta; \Gamma_1 \vdash R.l : T_1.l_k \vdash \Gamma_1 \forall 1 \leq k \leq m$$
Since $l$ is one of the fields of $R$, from (12) and (13) we get
\[
\Delta; \Gamma_1 \vdash h(a).l : T_1.l \vdash \Gamma_1 \quad (14)
\]
Let $\Gamma_3 = \Gamma_1$. From (a), (14), (9), (10) and T-State, we conclude
\[
\Delta; \Gamma_3 \vdash (h, h(a).l) : T \vdash \Gamma_2 \quad (15)
\]
**R-UnAssign.** Then $t = (o.l := o')$. From the premises and the shape of the conclusion of (b), we know
\[
\Delta; \Gamma_1 \vdash o' : C \vdash \Gamma_2 \quad (16)
\]
\[
\Delta; \Gamma_1 \vdash o.l : C \vdash \Gamma_2 \quad (17)
\]
\[
T = \text{Empty} \quad (18)
\]
From the premise of R-UnAssign, we further know $\text{ctype}(C)$ is a non-dependent type. Therefore, by (a), (15), (16) and heap update (Lemma 10), we obtain
\[
\Delta; \Gamma_1 \vdash h\{o.l \mapsto o'\} \quad (19)
\]
From assumption, we know $\Delta \vdash \Gamma_2$ and $o_{\text{Empty}} \in \Gamma_2$. Therefore, by T-Var, we obtain
\[
\Delta; \Gamma_2 \vdash o_{\text{Empty}} : \text{Empty} \vdash \Gamma_2 \quad (20)
\]
Let $\Gamma_3 = \Gamma_2$. By (19), (20), (17), (18) and T-State, we conclude
\[
\Delta; \Gamma_3 \vdash (h\{o.l \mapsto o'\}, o_{\text{Empty}}) : T \vdash \Gamma_2 \quad (21)
\]
**R-LinAssign.** Then $t = (o.l := o').l'$. This case is similar to the one above, except that $o'.l'$ has some dependent type $T_i$ and $\Gamma_2 = \Gamma_1\{o.l \mapsto T\}\{o'.l' \mapsto \text{Empty}\}$, which means that we take an additional step to derive $\Delta; \Gamma_2 \vdash h\{o.l \mapsto h(o').l'\}\{o'.l' \mapsto \text{Empty}\}$. By a similar argument to the one used to obtain (20), we conclude $\Delta; \Gamma_3 \vdash (h\{o.l \mapsto h(o').l'\}\{o'.l' \mapsto \text{Empty}\}, o_{\text{Empty}}) : T \vdash \Gamma_2$.

**R-NewAssign.** Then $t = o.l := (\text{new } C)\bar{o}$. From the premises and shape of the conclusion of (b), we know
\[
C.\text{init} = \Pi \Delta'.T_2 \times (\bar{U} \rightsquigarrow \bar{U}') \quad (22)
\]
\[
\Delta \vdash \theta : \Delta' \quad (23)
\]
\[
\Delta; \Gamma_1 \vdash \bar{o} : \bar{U}\theta \vdash \Gamma_1 \quad (24)
\]
\[
\Delta; \Gamma_1 \vdash o.l : T_1 \vdash \Gamma_1 \quad (25)
\]
\[
\Gamma_2 = \Gamma_1\{o.l \mapsto T_2\theta\} \quad (26)
\]
\[
T = \text{Empty} \quad (27)
\]
From the premises of R-NewAssign, we further know
\[
C.\text{fields} = \bar{l} \quad (28)
\]
\[
o' \text{ fresh} \quad (29)
\]
\[
h' = h, (o' = C\{\bar{l} = \bar{o}\}) \quad (30)
\]
By (29), we know that the fresh object \( o' \) is initialised, and by (21) and (22) we also know \( \Delta \vdash T_2 \theta \) and \( o' \) has type \( T_2 \theta \). So, from (a), (23) and \( \text{R-HeapObject} \), we get

\[
\Delta; \Gamma_2 \vdash h'
\]  

(30)

From (24) and weakening (Lemma 5), we also obtain

\[
\Delta; \Gamma_2 \vdash o.l : T_1 \vdash \Gamma_2
\]  

(31)

By (30), (31) and heap update (Lemma 10), we get

\[
\Delta; \Gamma_2 \vdash h' \{o.l \mapsto o'\}
\]  

(32)

From assumption, we know \( \Delta \vdash \Gamma_2 \) and \( o\text{Empty} \in \Gamma_2 \). Therefore, by \( \text{T-Var} \), we obtain

\[
\Delta; \Gamma_2 \vdash o\text{Empty} : \text{Empty} \vdash \Gamma_2
\]  

(33)

Let \( \Gamma_3 = \Gamma_2 \). By (32), (33) and \( \text{T-State} \), we conclude

\[
\Delta; \Gamma_3 \vdash (h' \{o.l \mapsto o'\}, o\text{Empty}) : T \vdash \Gamma_2
\]  

\( \text{R-Call} \). Then \( t = (o.l) o' \). From the premises and the shape of the conclusion of (b), we know

\[
\Delta; \Gamma_1 \vdash o : T'' \vdash \Gamma_1
\]  

(34)

\[
T''.l = \Pi \Delta' \cdot (T_1 \mapsto T_2) \times (U_1 \mapsto U_2)
\]  

(35)

\[
\Delta \vdash \theta : \Delta'
\]  

(36)

\[
\Delta \vdash T'' < : T_1 \theta
\]  

(37)

\[
\Delta; \Gamma_1 \vdash o' : U_1 \theta \vdash \Gamma_1
\]  

(38)

\[
T = \text{Empty}
\]  

(39)

\[
\Gamma_2 = \Gamma_1 \{o' \mapsto U_2 \theta\} \{o \mapsto T_2 \theta\}
\]  

(40)

From the premise of \( \text{R-Call} \), we further know

\[
l(y) = t' \in h(o).\text{class}
\]  

(41)

Since (41) implies \( (h, (o.l) o') \mapsto (h, t'[^{0/\theta}\text{tn}][o'/y]) \), we need to type this resulting state. From the assumption of well-formed programs, we know by (41) that one of the premises of \( \text{T-Record} \) must be

\[
\Delta_1, \text{this} : T'' \vdash l(y) = t'
\]  

(42)

which is the conclusion of \( \text{T-Method} \). We also know that (35) must be one of its premises. Considering this, we can derive

\[
\Delta' : \text{this} : T_1, y : U_1 \vdash t' : T \vdash \text{this} : T_2, y : U_2
\]  

(43)
From (36), (43), weakening (Lemma 4), followed by permutation (Lemma 1), we obtain

$$\Delta, \Delta'; \text{this} : T_1, y : U_1 \vdash t' : T \Downarrow \text{this} : T_2, y : U_2$$  \hspace{1cm} (44)

Using (36), (44) and substitution of index terms (Lemma 6), we derive

$$\Delta; \text{this} : T_1\theta, y : U_1\theta \vdash t' : T \Downarrow \text{this} : T_2\theta, y : U_2\theta$$  \hspace{1cm} (45)

We now apply Lemma 7 twice, for the substitution of $y$ and then of this. From (45) and (38), the first one yields

$$\Delta; \text{this} : T_1\theta, y : U_1\theta \vdash t'[^o'] : T \Downarrow \text{this} : T_2\theta, y : U_2\theta$$  \hspace{1cm} (46)

For the second substitution, we first apply $T$-Sub to (34) and (37) to get

$$\Delta; \Gamma_1 \vdash o : T_1 \dashv \Gamma_1$$  \hspace{1cm} (47)

Now, we can apply permutation (Lemma 2) twice, and one substitution using (46) and (47) in order to obtain

$$\Delta; o : T_1\theta, o' : U_1\theta \vdash t[^o'/y][^o'\,\theta] : T \Downarrow o : T_2\theta, o' : U_2\theta$$  \hspace{1cm} (48)

Let $\Gamma_3 = o : T_1\theta, o' : U_1\theta$. From (a), (48) and T-State, we conclude

$$\Delta; \Gamma_3 \vdash (h, t[^o'/y][^o'\,\theta]) : T \dashv \Gamma_2$$

**R-IfTrue.** Then $t = \text{if } (o = o') \text{ then } t_1 \text{ else } t_2$. From the premises of (b), we know

$$\Delta; \Gamma_1 \vdash o : T_1 \dashv \Gamma_1$$  \hspace{1cm} (49)

$$\Delta; \Gamma_1 \vdash t_1 : T \dashv \Gamma_2$$  \hspace{1cm} (50)

$$\Delta; \Gamma_1 \vdash t_2 : T \dashv \Gamma_2$$  \hspace{1cm} (51)

Let $\Gamma_3 = \Gamma_1$. Since $o = o$, we know that the resulting term is $t_1$ and so by (a), (50) and T-State, we conclude

$$\Delta; \Gamma_3 \vdash (h, t_1) : T \dashv \Gamma_2$$

**R-IfFalse.** This case is similar to the one above, except for $o \neq o'$ and the resulting term $t_2$.

**R-While.** Then $t = \text{while } (o = o') \text{ do } t'$. From assumption, the term reduces by R-While, rewritten to a nested conditional construct. So, we need to analyse the state $\Delta; \Gamma_3 \vdash (h, \text{if } o = o' \text{ then } (t'; \text{while } o = o' \text{ do } t') \text{ else } o_{\text{Empty}}) : T \dashv \Gamma_2$.

We have two cases to consider. When $o'$ is $o$, then we know

$$\Delta; \Gamma_3 \vdash o : T_1 \dashv \Gamma_3$$  \hspace{1cm} (52)

$$\Delta; \Gamma_3 \vdash t'; \text{while } o = o' \text{ do } t' : T \dashv \Gamma_2$$  \hspace{1cm} (53)

$$\Delta; \Gamma_3 \vdash o_{\text{Empty}} : T \dashv \Gamma_2$$  \hspace{1cm} (54)

$$\Gamma_2 = \Gamma_3$$  \hspace{1cm} (55)

$$T = \text{Empty}$$  \hspace{1cm} (56)
Since the resulting term is (53), by (a), (55), (56) and T-STATE, we conclude
\[ \Delta; \Gamma_3 \vdash (h, t'; \text{while } o = o' \text{ do } t') : T \vdash \Gamma_2 \]

When \( o \neq o' \), then the resulting term is (54) and by (a), (55), (56) and T-STATE, we conclude
\[ \Delta; \Gamma_3 \vdash (h, o_{\text{Empty}}) : T \vdash \Gamma_2 \]

**R-Seq.** Then \( t = o; t' \). From the premises of (b), we know
\[
\begin{align*}
\Delta; \Gamma_1 \vdash o & : T_1 \vdash \Gamma_1 \\
\Delta; \Gamma_1 \vdash t' & : T \vdash \Gamma_2 \\
\Gamma_3 = \Gamma_1 
\end{align*}
\]

From (a), (58), (59) and T-STATE, we conclude
\[ \Delta; \Gamma_3 \vdash (h, t') : T \vdash \Gamma_2 \]

**D Proof of Theorem 4, Progress**

We start with inversion, used in the proof of progress.

**Lemma 19 (Inversion of the Typing Relation).** Let \( \Delta; \Gamma_1 \vdash t : T \vdash \Gamma_2 \).

1. If \( t = y \), then \( \Gamma_2 = \Gamma_1 \text{ and, for some } T' \), we have \( \Delta \vdash \Gamma_1 \) and \( y : T' \in \Gamma_1 \) and \( \Delta \vdash T' <: T \).
2. If \( t = y \ell \), then \( \Gamma_2 = \Gamma_1 \) and, for some \( T' \), we have \( y : T' \in \Gamma_1 \) and \( \Delta \vdash T' \ell \in T \) and \( T \) and \( T' \) must have one of the forms \( C, \{i_k : T_k \}_{k=1}^n \), or \( T''i \).
3. If \( t = (y \ell := s) \), then there are two cases to consider. When for some \( C \)
\[ \Delta; \Gamma_1 \vdash y \ell : C \vdash \Gamma_1 \text{, then } \Delta; \Gamma_1 \vdash s : C \vdash \Gamma_1 \text{ and } \Gamma_2 = \Gamma_1 . \]
When \( \Delta; \Gamma_2 \vdash y \ell : T' \ell \vdash \Gamma_3 \), then for some \( U \) we have \( \Delta; \Gamma_1 \vdash s : U \vdash \Gamma_1 \text{ and } \Gamma_2 = \Gamma \{y \ell \mapsto U\} \).
4. If \( t = y \ell := (\text{new } C)s \), then \( \Pi \Delta'.T' \times U \dashv \bar{U}' \) and, for some substitution \( \theta \), we have \( \Delta \vdash \theta : \Delta' \) and \( \Delta; \Gamma_1 \vdash s : U \theta \vdash \Gamma_1 \) and \( \Delta; \Gamma_1 \vdash y \ell : U \vdash \Gamma_1 \) and \( T = \text{Empty and } \Gamma_2 = \Gamma_1 \{s \mapsto \ell \theta \} \{y \ell \mapsto T' \theta \} \).
5. If \( t = (s_1, l)s_2 \), then, for some type \( T' \) and substitution \( \theta \), we have \( \Delta; \Gamma_1 \vdash s_1 : T' \vdash \Gamma_1 \) and \( T', l \text{ is of the form } \Pi \Delta'.(T_1 \dashv T_2) \times (U_1 \dashv U_2) \) and \( \Delta \vdash \theta : \Delta' \) and \( \Delta \vdash T' <: T_1 \theta \) and \( \Delta; \Gamma_1 \vdash s_2 : U_1 \theta \vdash \Gamma_1 \) and \( T = \text{Empty and } \Gamma_2 = \Gamma_1 \{s_2 \mapsto U_2 \theta \} \{s_1 \mapsto T_2 \theta \} \).
6. If \( t = \text{if } s_1 = s_2 \text{ then } t_1 \text{ else } t_2 \), then for some \( T_1 \) and \( T_2 \), we have \( \Delta; \Gamma_1 \vdash s_1 : T_1 \vdash \Gamma_1 \) and \( \Delta; \Gamma_1 \vdash s_2 : T_2 \vdash \Gamma_1 \) and \( \Delta; \Gamma_1 \vdash t_1, t_2 : T_2 \vdash \Gamma_2 \) and \( \Delta \vdash T_2 <: T \).
7. If \( t = \text{while } s_1 = s_2 \text{ do } t \), then for some \( T_1 \) and \( T_2 \), we have \( \Delta; \Gamma_1 \vdash s_1, s_2 : T_1 \vdash \Gamma_1 \) and \( \Delta; \Gamma_1 \vdash t : T_2 \vdash \Gamma_2 \) and \( \Delta \vdash T_2 <: T \).
8. If \( t = t_1 ; t_2 \) then, for some type \( T_1 \) and \( T_2 \) and some \( \Gamma_3 \), we have \( \Delta; \Gamma_1 \vdash t_1 : T_1 \vdash \Gamma_3 \) and \( \Delta; \Gamma_3 \vdash t_2 : T_2 \vdash \Gamma_2 \) and \( \Delta \vdash T_2 \equiv T \).

**Proof.** By simple inspection of the typing rules. The final rule in any derivation concludes with \( \Delta; \Gamma_1 \vdash t : T \vdash \Gamma_2 \), using the “natural” typing of \( t \) and possibly one application of T-Sub to obtain that result whenever \( \Delta; \Gamma_1 \vdash t : T' \vdash \Gamma_2 \). This only holds because the subtyping relation is reflexive and transitive (a preorder).

**Theorem 4 (Progress).** Suppose \( \Delta; \Gamma_1 \vdash (h, t) : T \vdash \Gamma_2 \). Then, one of the following holds:

1. \( t \) is a value;
2. \( (h, t) \rightarrow (h', t') \).

**Proof.** By induction over the typing derivations. The hypothesis \( \Delta; \Gamma_1 \vdash (h, t) : T \vdash \Gamma_2 \) is the conclusion of rule T-State, whose premises are (a) \( \Delta; \Gamma_1 \vdash h \) and (b) \( \Delta; \Gamma_1 \vdash t : T \vdash \Gamma_2 \). We proceed by case analysis on the form of \( t \), using the appropriate clause of the inversion lemma (Lemma 19) on the typing relation, which leave us with eight cases to consider.

1. Case \( t = o \). The result is immediate, since \( t \) is a value.
2. Case \( t = o.l \). By the induction hypothesis, either \( t \) is a value or else it can take a reduction step. If \( t \) can take a step, then we have to prove \( h(o) = R \) and \( R.l \) is defined. By inversion on (b), we know \( T \) must have one of the three forms: \( C, \{l_k : T_k^{k \in 1...m+n}\} \), or \( T' l^i \), and we further have

\[
\begin{align*}
\Gamma_2 &= \Gamma_1 \\
o &: T' \in \Gamma_1 \\
\Delta \vdash T', l <: T
\end{align*}
\]

Take \( T' = \{l_k : T_k^{k \in 1...m+n}\} \) and \( R = C\{l_k = o_k^{k \in 1...m}\} \). From (a), (61) and agreement (Lemma 8), we get

\[
\begin{align*}
\Gamma_1 (o) &= \{l_k : T_k^{k \in 1...m+n}\} \\
h(o) &= C\{l_k = o_k^{k \in 1...m}\} \\
\Delta; \Gamma_1 \vdash o_k : T_k \vdash \Gamma_1, \forall 1 \leq k \leq m
\end{align*}
\]

By (62), we know \( l \) is one of the members of \( T' \), and by its shape it must be a field. Therefore, from (63) and (64), we have \( l = l_j \) and \( T = T_j \) for some \( 1 \leq j \leq m \). In particular, \( \Delta; \Gamma_1 \vdash h(o).l : T \vdash \Gamma_1 \) is one of the judgements defined in (65), so R-Field applies to \( t \).

3. Case \( t = (o.l := t_1) \). This case has three possible subcases. When \( t_1 = o' \), we can make a similar reasoning to the one above and conclude \( h(o') \) is defined. Since its class declares a non-dependent type, R-UNASSIGN applies to \( t \). When \( t = o'.l' \), we can make a similar reasoning, concluding \( h(o'), l' \) is defined. Since the object is an instance of a dependent class, R-LINASSIGN
applies to $t$. Finally, when $t = (\text{new } C)o$, then by definition $C$.fields means $l$, the fresh identifier is initialised and added to the heap. Therefore, R-NewAssign applies to $t$.

4. Case $t = (o.l)o'$. By inversion on (b), we know

\[
\begin{align*}
\Delta; \Gamma_1 \vdash o : T' \vdash \Gamma_1 & \quad (66) \\
T'.l & = I\Delta'.(T_1 \rightsquigarrow T_2) \times (U_1 \rightsquigarrow U_2) \quad (67) \\
\Delta \vdash \theta : \Delta' & \quad (68) \\
\Delta \vdash T' <: T_1\theta & \quad (69) \\
\Delta; \Gamma_1 \vdash o' : U_1\theta \vdash \Gamma_1 & \quad (70) \\
T & = \text{Empty} \quad (71) \\
\Gamma_2 & = \Gamma_1\{o' \mapsto U_2\theta\}\{o \mapsto T_2\theta\} \quad (72)
\end{align*}
\]

By (66), (69) and T-Sub, we obtain

\[
\Delta; \Gamma_1 \vdash o : T_1\theta \vdash \Gamma_1
\]

(73)

Using a similar argument to the one used in previous cases, from (73) we know $o \in \Gamma_1$. From this together with (a) and agreement (Lemma 8), we conclude $o \in h$. By (73), (67) and (68), we also know that the type of $o$ (with the substitution applied) matches the expected type in $T'.l$. Similarly, from (70) we can conclude $o' \in h$, noting that $o'$ (with the substitution applied) has the expected type in $T'.l$. Since R-CALL tell us $l(y) = t \in h(o).\text{class}$, the method body $t$ may be prepared to be further reduced by replacing the special variable this with receiver’s identifier $o$ and the formal parameter $y$ with the argument’s identifier $o'$ in the heap. Therefore, R-CALL applies to $t$.

5. Case $t = \text{if } (s_1 = s_2) \text{ then } t_1 \text{ else } t_2$. By the induction hypothesis, either $s_1$ is a value or else $s_1$ can take a reduction step, and similarly for $s_2$. If $s_1$ is $o$ or $s$ is $o'$, then either R-IfTrue or R-IfFalse applies to $s$. On the other hand

6. Case $t = \text{if } (o = o') \text{ then } t_1 \text{ else } t_2$ and $o \neq o'$. R-IfFalse applies to $t$.

7. Case $t = \text{while } (o = o') \text{ do } t$. R-While applies to $t$.

8. Case $t = (o; t)$. R-Seq applies to $t$. 

50