Freedom Before Commitment

Simple Flexible Initialisation for Non-Null Types

Alexander J. Summers and Peter Müller

ETH Zurich, Switzerland
{Alexander.Summers,Peter.Mueller}@inf.ethz.ch

Abstract. Null pointer dereferences are the most common runtime error in languages such as Java and C#. To alleviate this problem, Fähndrich and Leino proposed a *non-null type system*, in which reference types can be annotated with non-nullity expectations that are statically enforced. The main challenge for the static checking of non-nullity is object initialisation; since fields of new objects get first initialised with null, a non-null type system needs to determine when an object is fully initialised and, thus, its non-null fields actually contain non-null values. Several proposed solutions exist for tackling this problem, but each fail on one of the three criteria of soundness, flexibility and simplicity. In this paper we present a solution which satisfies all three criteria, making it suitable for mainstream use. We provide a formalisation of the core type system and prove soundness for a small language. We also show informally how the core type system can be extended to support a realistic language.

1 Introduction

Null pointer dereferences remain the most common source of runtime errors in object-oriented languages such as Java. The null value cannot easily be dispensed with in such languages; it permits an incremental initialisation of complex data structures (essential if such data structures are potentially cyclic), provides a default value for all reference types (essential in a type-safe language with heap references), and has many other common uses (such as representing "no such object" as the result of a lookup in a collection). Nonetheless, after their initialisation, the majority of variables in a program never contain null [2]. However, in mainstream languages the non-nullness assumption remains implicit or is documented informally such that compilers cannot enforce it. Accidental violations of the assumption (for instance, when passing null as an argument to a method that expects a non-null reference or when accessing a field before it has been initialised) are therefore not caught by the compiler and lead to runtime errors.

To detect runtime dereferences statically, Fähndrich and Leino proposed a *non-null type system* [6], in which reference types can be annotated with non-nullity expectations. For a class C, the type $C?$ indicates a reference which may possibly refer to the null value ("possibly-null C") , while the type $C!$ indicates a reference which is guaranteed to refer to an object ("non-null C"). Only variables

of non-null types may be dereferenced, which guarantees the absence of null-dereference errors at runtime. This idea has been widely adopted in the research community—various non-null type systems have been developed for Spec# (an extension of C#) [5, 11], Eiffel [13, 3, 10], and Java [4].

The main technical challenge in designing a non-null type system is how to handle object initialisation. The problem is that the runtime system initialises all fields of a new object with zero-equivalent values. So even fields declared as non-null start out being null. It is the obligation of the program, in particular, the constructor, to initialise these fields with non-null values. Until this initialisation is complete, it would not be sound to make use of declared non-null information about the fields of the newly-created object. However, this problem is not localised to constructor bodies because arbitrary code can be executed from inside a constructor. For example, when the newly-created object is passed as an argument to a method, that method must not assume the non-null declared fields to actually contain non-null values. Extra precautions are needed if a newly-created object can be assigned to the fields of another object; we must then consider the possibility of its fields being indirectly accessed via other references. Both situations are important in practice; for instance when two mutually recursive objects are created (say, a List and its sentinel Node), the constructor of the first object might pass the newly-created object to the constructor of the second object, where the two objects get linked to each other by mutual references.

Previous Work. Several solutions have been proposed for tackling the problem of initialisation for a non-null type system. We consider three main requirements that must be considered for any proposal aimed at broad use by programmers:

1. *Soundness:* The approach must guarantee the purpose of the non-null type system—that null dereference errors are prevented statically.
2. *Flexibility:* Common programming idioms must be supported, e.g., mutual initialisation of multiple objects, and initialisation of cyclic data structures.
3. *Simplicity:* The required annotations must be few and understandable by programmers.

The original work of Fähndrich and Leino [6] introduced *raw types* to handle initialisation; In addition to the non-null information, raw types have an additional annotation indicating that the referred object may not be fully initialised and, thus, may not be reliable in terms of non-null guarantees. A newly-created object in a constructor is naturally typed as raw, and raw references can only be used in positions where they are explicitly declared to be permitted. In particular, the system does not permit a raw reference to be assigned to a field of any object, even of the referenced object itself. This restriction makes the solution inflexible; in particular, it prevents common implementations such as the mutual initialisation of multiple objects, or cyclic data structures. Ekman and Hedin's system [4] implements raw types for Java and adds type inference.

Fähndrich and Xia introduced *delayed types* [5]. Delayed types decorate reference types with a *delay time* which indicates the notional point during execution

after which the referenced object satisfies its non-null annotations. *Delay scopes* are introduced into the program text to indicate points at which certain times will expire. Delay times on reference types can be existentially quantified, with bounds expressing relationships between various delay times. Because references to many objects can share the same delay types, the system is flexible enough to support practical examples. However, the complexity of the type and program annotations makes the presented system unfeasible for use at the source language level. Indeed, when implementing the system in Spec# [11], it was decided to greatly cut down the complexity of the type system, including only a single extra "Delayed" attribute in the language, representing an unknown delay time. The resulting implementation is however unsound: at method calls it allows any parameters to be provided as delayed arguments, but inside the method bodies assumes each such argument to have the same delay time. Fixing this problem by enforcing that all delayed references have the same delay time would make the system too inflexible to handle the mutual initialisation of multiple objects.

Eiffel's non-null types (called "attached types") do not appear to address the problem of object initialisation soundly. According to the Eiffel standard [3], a field (variable attribute) of class C may be considered *properly set* (essentially, fully initialised) provided it "... is (recursively) properly set at the end position of every creation procedure of C." Because creation procedures (constructors) can themselves contain arbitrary code, this is not sufficient for soundness. The problematic situation can sometimes be avoided by providing default creation procedures for all types of attached variable attributes (non-null fields)—these get implicitly called when a field is found not to be properly initialised yet. However, default initialisation cannot handle cases such as cyclic lists, or the mutual initialisation of objects. The actual Eiffel implementation appears (by experiment) to actually prevent unsoundness by enforcing a much stronger rule: an object under construction may not be used as receiver or argument of any call, or be assigned to any field until its initialisation is complete. This rule makes cyclic and mutual initialisations impossible.

The recent work of Qi and Myers [14] proposes *masked types* to tackle object initialisation. This system provides versions of class types in which any subset of fields can be "masked", indicating that the initialisation of such fields cannot be relied upon. This permits various kinds of incremental initialisation, including cyclic structures. However, even the simple examples found in their paper require many annotations. So while this system seems to be the most flexible approach yet, it is unlikely that an average programmer would find it usable to handle the everyday problem of sound object initialisation. We give a more detailed explanation of this view and comparison of our system with both Delayed Types and Masked Types, in Section 4.

In summary, each of the existing solutions fails on one of the three requirements we consider essential for the usefulness of a non-null type system:

Unsound	**Inflexible**	**Complex**
Spec# implementation	Raw Types	Delayed Types paper
Attached Types papers	Eiffel implementation	Masked Types

Contributions. The contribution of this paper is the design of a non-null type system that satisfies the three requirements above. The concrete technical contributions are (1) a non-null type system in which object initialisation is handled soundly, flexibly, and with low annotation overhead, and (2) a formalisation of the type system for a small language, and proof of preservation. To our knowledge this is the first formal proof for a type system dealing with object initialisation for non-null types.

We present the design of our type system informally in Sec. 2. The formalisation is then presented in Sec. 3. We show some detailed examples and comparisons with related work in Sec. 4. Finally, we discuss some further related work (Sec. 5) and conclude (Sec. 6).

2 The Design

The innovations of our type system are all concerned with object initialisation. For the basic non-nullity, we adopt the same distinction as is made in all existing approaches: each reference type C (in the declaration of a field, variable, method signature, or in a cast, etc.) is replaced by two variants $C?$ and $C!$, indicating a possibly-null and a non-null type, respectively. Null pointer exceptions are avoided by forbidding the dereferencing of a reference with possibly-null type[1]. $C!$ is a subtype of $C?$ for any C since $C!$ is a specialisation of $C?$ both in terms of sets of possible values and in terms of behavior (one can do strictly less with a $C?$ reference). With this subtype relation, the usual type rule for assignment ensures that only non-null values can be assigned to variables declared with a non-null type (called *non-null variables* in the following). In particular, it ensures that the initialisation of non-null fields is *monotonic*: once a non-null field has been initialised with a non-null value, it will remain initialised. Therefore, one need only be careful that the non-null declarations cannot be relied upon before initialisation takes place. We will explain the machinery for object initialisation in the following and illustrate it using the example in Fig. 1 (the motivating example from [6]).

2.1 Initialisation Expectations

We prevent a program from relying on the non-nullity information for an uninitialised field by tracking in the type system whether an object is expected to be fully initialised or not. Like with raw types and delayed types, non-nullity information may be assumed only for fields of objects that can be expected to be initialised, and references stored in an object that is expected to be initialised must point to objects that are also initialised (that is, the initialisation guarantees are "deep"). However, our system differs from previous work in the handling

[1] A dataflow analysis can be used to allow such dereferencing *after* checking at runtime that the reference is in fact not `null`, for instance using an if-statement. Such a dataflow analysis is important for the practicality of the type system, but it is orthogonal to the focus of this paper and therefore ignored in the following.

```
class List {
  Node! sentinel ;
  List() { this.sentinel = new Node(this); }

  void insert (Object? data) {
    this.sentinel.insertAfter(data);
  }
}

class Node {
  List! parent; Node! prev; Node! next;
  Object? data;
  // for sentinel construction
  Node([Free] List! parent) {
    this.parent = parent;
    this.prev = this;
    this.next = this;
  }
  // for data node construction
  Node(Node! prev, Node! next, Object? data) {
    this.parent = prev.parent ;
    this.prev = prev;
    this.next = next;
    this.data = data;
  }
  void insertAfter (Object? data) {
    Node newNode = new Node(this, this.next, data);
    this.next.prev = newNode;
    this.next = newNode;
  }
}
```

Fig. 1. Doubly-linked list example. The List constructor illustrates mutual object initialisation; the this reference is passed to the first Node constructor and assigned to the node's parent field while the List object is still under initialisation. Like in Java's LinkedList implementation, the nodes of our list form a cyclic structure, whose initialisation is illustrated by the first Node constructor. The [Free] annotation in its signature is explained in the text.

of objects that are still under initialisation. This difference is illustrated by the example in Fig. 2.

A non-null type system must reject the constructor of class C because its execution leads to a null dereference exception. The constructor first initialises field f with a reference to the (already initialised) object p. The next statement is the one that causes the problem: it stores the this reference in a field of the initialised object p, which violates the deep initialisation guarantee of p. This violation is then exploited in the third statement by expecting falsely that all

```
public class C
{
  C! f, g;

  public setF(C! q) { this.f = q; }

  public C(C! p) {      // "this" is not initialised, but p is
    this.setF(p);       // alias p via field of "this"
    this.f.setF(this);  // assign this to p.f, so p.f.g is null
    this.g = p.f.g.f;   // null dereference exception
  }
}
```

Fig. 2. Example of faulty object initialisation.

objects reachable from p are initialised and, thus, their non-null fields contain non-null values, which is not the case for this.g.

Raw types prevent this example by forbidding raw references to be stored in any field. So if setF's parameter q is typed as raw, the method body does not type check. If q is not raw then the call this.f.setF(this) does not type check because this is raw inside the constructor. However, while this solution is type-safe, it prevents implementations such as the first Node constructor in Fig. 1, which assigns objects that are still under initialisation to all three fields. Delayed types prevent the faulty example essentially by requiring of the call this.f.setF(this) that this and this.f (that is, p) have the same delay time, which is not the case because p is initialised, but this is not. We already argued in the introduction that this treatment is sound, but makes the system complex. The simplified version of delayed types implemented in Spec# does not prevent the example, which illustrates that this system is unsound! If both the receiver and the parameter of setF are marked as delayed, the type system assumes that both have the same delay time and permits the assignment. However, this assumption is not (and cannot) be checked at the call site, which causes the unsoundness.

The core challenge illustrated by this example is how to allow storing an object that is still under initialisation in a field of another object without incurring the overhead of tracking their delay times. Such an assignment is safe if we know that the field belongs to an object that is *definitely* considered to be still under initialisation and unsafe otherwise. However, this definite information is not available in raw types or in Spec#'s version of delayed types because in these systems, raw or delayed references may point to initialised objects or objects under construction (that is, raw and delayed types are supertypes of non-raw and non-delayed types, respectively). This is what allows the faulty constructor to pass the initialised p and the not-yet-initialised this reference to the setF method when its parameter is declared raw or delayed.

We solve this challenge by tracking in our type system whether an object is *expected to be initialised* or *definitely not expected to be initialised*. We say that

an object is *locally initialised* if every one of its non-null fields contains a non-null value; an object is *deeply initialised* if every object reachable from the object is locally initialised. We can then distinguish three kinds of reference types:

1. *Committed references* point to objects that are guaranteed to be deeply initialised.
2. *Free references* point to objects that are guaranteed *not to be reachable* from *any* committed reference. No information is guaranteed about the values stored in the fields of the referred-to objects. That is, the object referred to by a free reference may or may not be initialised. The crucial point is that it is not *expected* to be initialised; so it is "free" of obligations.
3. *Unclassified references* may point to any object. This kind is a supertype of the corresponding committed and free kinds.

Our committed kind corresponds to non-raw and non-delayed types in previous systems. The unclassified kind corresponds to raw types and Spec#'s delayed types as it subsumes objects that are expected to be initialised and objects that are not. The free kind is new; free references are allowed to have anything assigned to their fields, which permits flexible initialisation between multiple such objects. To make this safe we enforce that a free reference never aliases an object reachable from a committed reference.

Note that these initialisation expectations are independent of the non-nullity of a reference—we can have both non-null and possibly-null references of any of the three kinds above (in the latter case, guarantees about the "referred-to object" only apply if the reference is not null). Despite attaching both nullity information and an initialisation expectation to reference types, the annotation overhead of our system is quite low. Almost all references handled in a program are committed, non-null references, such that a suitable default avoids overhead for those references. Initialisation expectations need to be declared explicitly only for non-trivial initialisation patterns. In our examples, we make nullity information explicit; the default initialisation expectation for all reference types is committed, except for the type of this inside a constructor, which is free. We use the syntax [Free] and [Unclassified] to declare free and unclassified types, resp. With these defaults, the List example in Fig. 1 requires a single [Free] annotation.

2.2 Fields

Field types include non-null annotations, but we do not allow field types to carry initialisation expectations (e.g., there are no "free fields"); this is because our initialisation kinds make guarantees about a references at a given point in the execution. Since the initialisation expectation of a reference changes during execution (in particular, from free to committed when the initialisation is finished, see the next subsection), it would not be safe to store aliases with different initialisation kinds in fields.

Field Read. When reading a field $x.f$, we infer the nullity and initialisation expectation of the result as follows: The result is non-null if and only if f is declared non-null *and* x is committed (recall that the committed kind is the only kind that guarantees that the referenced object is initialised). The result is committed if and only if x is committed (since commitment provides a guarantee about all reachable objects); otherwise the result is unclassified since the fields of free references may alias both free and committed references.

Field Update. We now consider field updates of the form $x.f := y$. The nullity checks are trivial: x must be non-null, and the nullity of y must conform to the nullity declared for f. For the initialisation expectation, the update is allowed if the initialisation kinds satisfy at least one of the following two cases. First, if x is free, we may store objects with any initialisation expectation in its fields. This is acceptable because the free kind does not make any guarantees about the initialisation expectations of reachable objects. This case applies, for instance, to all field updates in the Node constructors of our List example (Fig. 1) because this is implicitly free inside a constructor. Second, if y is committed, we may assign it to fields of any object. If x is committed, then we preserve the deep initialisation guarantee; if x is free or unclassified, it does not make any guarantees about the initialisation expectations of reachable objects. In particular, we cannot make free references "more reachable" by such a field update, since they are guaranteed unreachable from the committed y in the first place.

Example. Let's now discuss how our system prevents the faulty example from Fig. 2. Consider the second call to setF in C's constructor. The receiver of this call, this.f, is unclassified because this is not committed. Therefore, the call type checks only if setF's receiver is declared unclassified. The argument of the call, this, can be typed with a free or unclassified kind. So the call type checks only if setF's parameter q is declared free or unclassified. In both cases, the field update in setF's body does not satisfy either of the two cases above (the receiver of the update is not free and the right-hand side is not committed) and is, thus, rejected by the type checker. This illustrates that our system prevents storing objects that are not expected to be initialised in fields of objects that are expected to be initialised, which prevents the unsoundness.

2.3 Constructors

While the use of free references allows our solution flexibility, the goal of object initialisation is that objects can eventually be expected to be deeply initialised such that one can rely on the non-null annotations of their fields. Since constructors are primarily responsible for initialising objects, initialisation expectations may change when a constructor terminates. In this subsection, we explain the details of this change of expectations. We do not consider implicit calls to supertype constructors here, but an extension is possible.

Local Initialisation. As a first step, we require that all constructors guarantee local initialisation of the created object, on their return. This is enforced statically using a straightforward *definite assignment analysis*, which checks that each non-null field of an object can be statically guaranteed to be assigned to at least once in its constructor body.

Whether or not the new object can be expected to be *deeply* initialised and whether it can be referred to by committed references depends on the values that get assigned to its fields, as we discuss in the following.

Constructor Calls with Free or Unclassified Arguments. Let us first consider constructor calls with at least one free or unclassified argument. If a free reference is passed as an argument a constructor, then the constructor may initialise fields of the newly-created object using the free reference. For example, the List constructor in Fig. 1 calls the first Node constructor and passes the free reference this as argument. The Node constructor stores this free reference in the parent field of the new Node object. So when the Node constructor terminates, the new Node n is not deeply initialised; in particular, n.parent.sentinel is still null. To consider the reference to n as committed would therefore be unsafe. The same argument applies if an unclassified reference is passed as argument to a constructor, since it may (via subtyping) disguise a free reference.

Considering the reference to the new object as free is safe if we can guarantee that the object is not reachable from any committed object. This is the case because inside the constructor the object was referred to via a free reference and, therefore, the constructor could not store the reference in the field of a committed object.

This rule for constructor calls illustrates that our initialisation expectations express whether the referenced object is *expected* to be initialised, not whether it is *known* to be initialised. When passing a free or unclassified reference to a constructor, we do not know statically whether the new object is deeply initialised upon termination of the constructor or not. However, this uncertainty does *not* force us to give the reference an unclassified kind. We can give a stronger guarantee because we know that no committed references reaches the new object, which is therefore not expected to be initialised.

Constructor Calls with Committed Arguments. Let us now consider constructor calls where all arguments are committed (which subsumes the case that the constructor does not have parameters). When such a constructor terminates, the new object n can safely be expected to be deeply initialised and to not reach any objects referred to by a free reference. So the creation expression yields a committed reference. For instance, when the List constructor in Fig. 1 terminates, the new List object n is deeply initialised (n and its sentinel node are locally initialised) and neither n nor its sentinel node are referred to by a free reference.

The intuitive justification for this rule is as follows. Consider the execution of the constructor of a new object n. During this execution, the set of reachable

objects consists of the set of objects R that are reachable from the constructor's arguments and the set of objects N that includes n and all objects created during the execution of n's constructor. For the List constructor, the set R is empty, whereas N contains the new List object and its sentinel node. When the constructor terminates, we know that all objects in N are locally initialised; the values assigned to their fields are references to objects in R or N (because these are all the reachable objects). At this point, we may regard all references to the objects in N as committed; these objects are deeply initialised because all objects reachable from them are in R or in N and thus locally initialised[2].

Regarding the objects in N as committed would be unsafe if the caller of the constructor could have a direct free reference to an object reachable from an object in N; this would violate the guarantees for free references. However, this cannot happen for the following reason. All objects reachable from objects in N are in R or N. Since all objects in R are reachable via a committed reference (the argument to the constructor), the caller cannot have a direct free reference to one of them. Since the objects in N do not exist before the constructor call, the caller's only direct reference to an object in N is the reference yielded by the creation expression, which is not free.

2.4 Casts and Runtime Support

As in other non-null type systems, we allow the down-casting of an expression from a possibly-null type to a non-null type. The associated runtime check ensures that the expression indeed evaluates to a non-null value. We do not however allow down-casting a reference type's initialisation expectation via casts (from unclassified to free or committed). This is primarily because these would have to be unchecked casts; in particular, the requirement that objects directly referred to by free references are never reachable from committed references, cannot be checked efficiently at runtime. Therefore, initialisation expectations need not be represented at runtime. Consequently, the only runtime support our system requires is simple non-null checks for down-casts to non-null types.

2.5 Generic Initialisation Expectations

The design presented so far can handle many examples including the common initialisation patterns. However, there are potentially useful methods that require a more expressive system. For example, consider an identity method, which simply returns the reference passed to it:

```
Object! id(Object! o) { return o; }
```

[2] This argument generalizes trivially to global data if the global variables (e.g., static fields) are enforced to store committed references.

This method should be callable with arguments with *any* initialisation expectation and yield a reference with the *same* initialisation expectation as the argument. This is not possible in the system introduced so far. Passing arguments with any initialisation expectation is permitted only if we type the formal parameter `o` as unclassified. But this forces us to make the result type also unclassified and so the result of a call to `id` generally does not have the same initialisation expectation as the actual argument reference.

To avoid having to write various versions of the `id` method for the possible initialisation expectations of `o`, we introduce a very simple form of genericity: Types of method parameters and results may be generic w.r.t. their initialisation expectations. In our formalisation we write $\texttt{Object}^{\alpha}\texttt{! id(Object}^{\alpha}\texttt{! o)}$ to express the desired method signature, in which α is an *expectation variable*. But notice that α is only a place-holder for one of three concrete initialisation expectations; this polymorphism is far simpler than that already typical in full Java or C#. In particular, we can type check the body of such a method by simply type checking it once per possible value of α. Therefore, our type system need not handle these generics explicitly, but we have to check each method body a number of times which is exponential in the number of distinct expectation variables in its signature. In practice this does not seem a serious issue since expectation variables are not needed for most examples, and even when they are, the need for several distinct variables at once seems unlikely.

The following *copy* method between two objects (assuming class `C` has some field `f`) illustrates an extra tweak to this feature:

```
void copy(C! x, C! y) { x.f = y.f; }
```

This method type checks according to our rules for field assignment if `x` and `y` either both have free types, or both have committed types. However, it does *not* type check in the case where `x` and `y` are both unclassified because two unclassified types do not actually express that both references have the same initialisation expectation—it might be the case that one reference aliases a committed reference and the other a free reference. In fact, in general we observe that insisting that two method arguments have the same initialisation expectation is useful *only* if that initialisation expectation is either free or committed. This observation leads us to slightly tweak our design for generic expectation variables: if such a variable occurs more than once in the parameter types (including the receiver expectation), then we do not permit instantiating that variable as "unclassified". This case is neither considered when type checking the method body nor permitted in calls. This tweak allows us to type our `copy` method above with the signature $\texttt{void copy(C}^{\alpha}\texttt{! x, C}^{\alpha}\texttt{! y)}$.

3 The Formalisation

In this section, we present a formalisation of our approach. Much of the formalisation is standard. However, the soundness arguments for our system are subtle,

especially the treatment of constructor calls described in Sec. 2.3. Our soundness results (in Sec. 3.4) make these arguments explicit.

3.1 Programming Language

We focus on a very simple language, which nonetheless illustrates the main features of the problems of object initialisation and our solutions. We consider a simple class-based language (without generics), in which we have exactly one constructor per class. Note that we do not model calls to supertype constructors here—a constructor is obliged to fully initialise a new object.

Definition 1 (Classes and Types). *We assume a finite set of* classes*, ranged over by C, D, and a pre-defined reflexive, transitive, acyclic* subclassing *relation on classes, written $C \leq D$.*
We assume a set of method names*, ranged over by m, and a set of* field names*, ranged over by f, g. We assume the existence of a function fields() from classes to sets of field names, and a function methods() from classes to sets of method names. These functions are monotonic increasing with respect to subclassing (i.e., applying the functions to subclasses cannot yield smaller sets).*
Non-null annotations*, ranged over by n, are defined by* $n ::= ? \mid !$
Initialisation expectations*, ranged over by k, are defined by:*

$$\begin{array}{rll} k ::= & 0 & \textit{(free)} \\ \mid & 1 & \textit{(committed)} \\ \mid & \diamond & \textit{(unclassified)} \end{array}$$

We assume a set of expectation variables*, ranged over by α.*
Generic initialisation expectations*, ranged over by γ, are defined by* $\gamma ::= k \mid \alpha$.
Types*, ranged over by T, are defined by* $T ::= C^k n$.
Signature Types*, ranged over by S, are defined by* $S ::= C^\gamma n$.
Simple Types*, ranged over by t, are defined by* $t ::= C\ n$.

For example, $C^0!$ is a type for a non-null free reference of class C, while $C^\diamond?$ is a type for a possibly-null unclassified reference. We employ signature types only on method declarations—expectation variables are always instantiated before use in our type system. Simple types are used in field declarations and in casts, where initialisation expectations are not permitted.

Subtyping combines specialisation of expectations, non-nullity and classes themselves (i.e., subclassing):

Definition 2 (Type Relations). Expectation specialisation *is a binary relation on initialisation expectations, written $k_1 \leq k_2$ and defined by: $k_1 \leq k_2 \Leftrightarrow k_1 = k_2 \vee k_2 = \diamond$.* Generic initialisation specialisation *is defined similarly by: $\gamma_1 \leq \gamma_2 \Leftrightarrow \gamma_1 = \gamma_2 \vee \gamma_2 = \diamond$.*
Non-null specialisation *is a binary relation on non-null annotations, written $n_1 \leq n_2$ and defined by: $n_1 \leq n_2 \Leftrightarrow n_1 = n_2 \vee n_2 = ?$.*
Subtyping *is a binary relation on types, written $T_1 \leq T_2$ and defined by:*
$C_1{}^{k_1} n_1 \leq C_2{}^{k_2} n_2 \Leftrightarrow C_1 \leq C_2 \wedge k_1 \leq k_2 \wedge n_1 \leq n_2$.

Subtyping projects down to simple types in the obvious way; we define:
$C_1\ n_1 \leq C_2\ n_2 \Leftrightarrow {C_1}^1 n_1 \leq {C_2}^1 n_2$
Subtyping is defined analogously for signature types :
${C_1}^{\gamma_1} n_1 \leq {C_2}^{\gamma_2} n_2 \Leftrightarrow C_1 \leq C_2 \wedge \gamma_1 \leq \gamma_2 \wedge n_1 \leq n_2$.
We define three auxiliary predicates on types. nullable$(C^k n)$ *holds exactly when* $n =?$. *committed*$(C^k n)$ *holds exactly when* $k = 1$. *free*$(C^k n)$ *holds exactly when* $k = 0$.

In order to define the type system and operational semantics, we require the existence of field and method lookup functions. In particular, we need to be able to retrieve the declared (simple) type for a field in a class, and the signatures of methods and constructors. Method signatures include the possibility of specifying an initialisation expectation for the *receiver* of the method call, as well as its arguments and return type. Constructor signatures do not have these two features; during execution of a constructor it's receiver is always a free reference, and after execution its initialisation expectation is determined by those of the passed arguments (cf. Subsection 2.3). Both kinds of signatures also include declarations of local variables used within the method body.

Definition 3 (Field and Method Lookups). Field type lookup *is modelled by a partial function fType*(C, f) *from pairs of class-name and field-name to simple types. It satisfies the restrictions that* $f \in fields(C) \Leftrightarrow (C, f) \in dom(fType)$ *and* $f \in fields(C)\ \wedge\ D \leq C \Rightarrow fType(D, f) = fType(C, f)$.
A Method Signature *is a four-tuple* $(\gamma, \overrightarrow{x_i{:}S_i}, S, \overrightarrow{y_j{:}S_j})$, *whose elements are: (1) a generic expectation annotation* γ, *indicating the initialisation expectation of the receiver, (2) a sequence* $\overrightarrow{x_i{:}S_i}$ *of parameter names (variable names) along with their declared Signature Types, (3) a signature type* S *representing the return value of the method;* S *may only mention expectation variables which occur in* γ *or parameter types, and (4) a sequence* $\overrightarrow{y_j{:}S_j}$ *of local variable names along with their declared Signature Types; these may only mention expectation variables which occur in* γ *or parameter types. A* Constructor Signature *is a two-tuple* $(\overrightarrow{x_i{:}S_i}, \overrightarrow{y_j{:}S_j})$, *similarly declaring parameters and local variables.*
Method signature lookup *is modelled by a partial function mSig*(C, m) *from pairs of class-name and method-name to method signatures. It satisfies the restriction that* $m \in methods(C) \Rightarrow (C, m) \in dom(mSig)$, *and the usual variance requirements for subclassing (covariant return types and contravariant parameter types).*
Method body lookup *is modelled by a partial function mBody*(C, m) *(with the same domain as mSig) from pairs of class-name and method-name to statements.*
Constructor signature lookup *is modelled by a function cSig*(C) *from class-names to constructor signatures.* Constructor body lookup *is modelled by a function cBody*(C) *from class-names to statements.*

While we allow generic initialisation expectations in method signatures, these are always instantiated with concrete initialisation expectations at each method call. When type checking a method body, we will type-check all possible such instantiations of the declared method signature.

Definition 4 (Generic Expectation Instances). *A* expectation substitution *θ is a partial function from expectation variables to expectation annotations. We write $\theta(\alpha)$ for its application to a particular expectation variable α, and extend this application to signature types (written $\theta(S)$) in the obvious way.*
For any sequence $\overrightarrow{S_i}$ of signature types, we write $vars(\overrightarrow{S_i})$ to denote the set of expectation variables occurring in $\overrightarrow{S_i}$. Note that if $vars(\overrightarrow{S_i}) \subseteq dom(\theta)$ then $\overrightarrow{\theta(S_i)}$ is a (well-defined) sequence of reference types.
An expectation substitution θ is an instantiation *for a sequence of signature types $\overrightarrow{S_i}$, written $instance(\theta, \{\overrightarrow{S_i}\})$, if it maps at least the variables occurring in the sequence of types, and does not map those variables occurring more than once to $\diamond$:*

$$instance(\theta, \{\overrightarrow{C_i{}^{\gamma_i} n_i}\}) \Leftrightarrow dom(\theta) \supseteq vars(\overrightarrow{C_i{}^{\gamma_i} n_i}) \wedge (i \neq j \wedge \gamma_i = \gamma_j = \alpha \Rightarrow \theta(\alpha) \neq \diamond)$$

For a sequence of signature types $\overrightarrow{S_i}$, the set of minimal instances *is written $instances(\overrightarrow{S_i})$ and defined by: $instances(\overrightarrow{S_i}) = \{\theta \mid dom(\theta) = vars(\overrightarrow{C_i{}^{\gamma_i} n_i}) \wedge instance(\theta, \{\overrightarrow{S_i}\})\}$*

Our statements include assignments, method and constructor calls and casts. We do not include conditionals since they would only be of interest when combining our type system with a dataflow analysis. Note that we do not have a return statement; methods return the value of a pre-defined local variable res. For simplicity, we treat field assignments, calls, object creation, and casts as statements. Complex expression can be decomposed using local variables.

Definition 5 (Expressions and Statements). *We assume a set of* program variables*, ranged over by x, y, z, including a distinguished variable* this*. Expressions, ranged over by e, are defined by the following grammar:*

$$e ::= x \mid x.f \mid \mathsf{null}$$

Statements, ranged over by s, are defined by the following grammar, with the extra restriction that (in all cases) x may not be the special variable this*:*

$$\begin{array}{rll} s ::= & x := e & \textit{(variable assignment)} \\ \mid & z.f := y & \textit{(field assignment)} \\ \mid & x := y.m(\overrightarrow{z_i}) & \textit{(method call)} \\ \mid & x := \mathsf{new}\ C(\overrightarrow{z_i}) & \textit{(object creation)} \\ \mid & x := (t)y & \textit{(cast)} \\ \mid & s_1; s_2 & \textit{(sequential composition)} \end{array}$$

Note that casts employ only simple types. As discussed in Section 2.4, we do not support casts that change the initialisation expectation of a reference.

3.2 Type System

We now turn to the definition of our type system, which includes *definite assignment* checks. There are two kinds of checks made. Firstly, a set Δ of definitely

assigned program variables is carried in judgements, to (conservatively) track which variables can be safely read from. In particular, non-null local variables can only be safely read from if they are named in the set Δ; this is necessary since all local variables are initialised to null in our operational semantics, regardless of their types. When typing expressions, we enforce this check—an expression is only well-typed if it reads only from variables named in the current Δ. When typing statements we use a "before" and "after" Δ to track this information in the type system.

Secondly, we employ a set of field names Σ, which conservatively record which fields of the current receiver are definitely assigned during execution of a statement. This set is relevant only for constructors; it is used to enforce the requirement that constructors guarantee to assign all non-null fields. Since we do not need to make any intermediate checks based on this set, it only occurs once in each typing judgement, indicating the fields definitely assigned between the beginning and end of execution of the statement.

Definition 6 (Static Type Assignment). *A* type environment Γ *is a partial function from program variables to reference types. An* assigned variables set Δ *is a set of variable names (indicating which have been definitely assigned). An* assigned fields set Σ *is a set of field names (indicating which fields of the receiver have been definitely assigned).*
Expression typing *is defined by judgements* $\Gamma; \Delta \vdash e : T$*, indicating that* e *has type* T *under assumptions* Γ*, and is safe to read from variables in* Δ*. The judgements are defined in Fig. 3.*
Statement typing *is defined by judgements* $\Gamma; \Delta \vdash s \mid \Delta'; \Sigma$*, indicating that* s *is well-typed under assumptions* Γ*, reads ony variables in* Δ *and, after execution, will guarantee that variables* Δ' *and fields* Σ *are definitely assigned. The judgements are defined in Fig. 4.*

$$\frac{\mathit{nullable}(x) \vee x \in \Delta}{\Gamma; \Delta \vdash x : \Gamma(x)}\ (\textsc{Tvar}) \qquad \frac{}{\Gamma; \Delta \vdash \mathit{null} : C^k?}\ (\textsc{Tnull})$$

$$\frac{\Gamma; \Delta \vdash x : C^{k_1}! \quad \mathit{fType}(C, f) = D\ n_1 \qquad k_2 = \begin{cases} 1 & \text{if } k_1 = 1 \\ \diamond & \text{otherwise} \end{cases} \qquad n_2 = \begin{cases} ! & \text{if } n_1 =! \text{ and } k_1 = 1 \\ ? & \text{otherwise} \end{cases}}{\Gamma; \Delta \vdash x.f : D^{k_2} n_2}\ (\textsc{Tfld})$$

Fig. 3. Expression typing.

We can now define what we type-check for class definitions - essentially each constructor and method definition must be well-typed with respect to each of the valid instances of its declared signature.

Definition 7 (Well-formed program). *For each class* C *of the program, we check that each method* $m \in \mathit{methods}(C)$ *is well-formed* ($\vdash_m C, m$)*, and that its*

$$\frac{\Gamma;\Delta \vdash e : T \quad T \leq \Gamma(x)}{\Gamma;\Delta \vdash x := e \mid \Delta \cup \{x\};\emptyset}\ (\textsc{TVarAss})$$

$$\frac{\Gamma;\Delta \vdash x : C^{k_1}! \quad \mathit{fType}(C,f) = D\ n \quad \Gamma;\Delta \vdash y : T \quad T \leq D^{k_2} n \quad k_1{=}0 \vee k_2{=}1 \quad \Sigma = \begin{cases} \{f\} & \mathit{if}\ x = \mathsf{this} \\ \emptyset & \mathit{otherwise} \end{cases}}{\Gamma;\Delta \vdash x.f := y \mid \Delta;\Sigma}\ (\textsc{TFldAss})$$

$$\frac{\Gamma;\Delta \vdash y : C^k! \quad \mathit{mSig}(C,m) = (\gamma, \overrightarrow{x_i{:}S_i}, S, \overrightarrow{y_j{:}S_j}) \quad \mathit{instance}(\theta, \{\overrightarrow{S_i}, S, C^\gamma!\}) \quad \overrightarrow{\Gamma;\Delta \vdash z_i : T_i} \quad \overrightarrow{T_i \leq \theta(S_i)} \quad \theta(S) \leq \Gamma(x) \quad k \leq \theta(\gamma)}{\Gamma;\Delta \vdash x := y.m(\overrightarrow{z_i}) \mid \Delta \cup \{x\};\emptyset}\ (\textsc{TCall})$$

$$\frac{\mathit{cSig}(C) = (\overrightarrow{x_i{:}S_i}, \overrightarrow{y_j{:}S_j}) \quad \mathit{instance}(\theta, \{\overrightarrow{S_i}\}) \quad \overrightarrow{\Gamma;\Delta \vdash z_i : T_i} \quad \overrightarrow{T_i \leq \theta(S_i)} \quad k = \bigwedge \mathit{committed}(T_i) \quad C^k! \leq \Gamma(x)}{\Gamma;\Delta \vdash x := \mathsf{new}\ C(\overrightarrow{z_i}) \mid \Delta \cup \{x\};\emptyset}\ (\textsc{TCreate})$$

$$\frac{\Gamma;\Delta \vdash y : C^k n_1 \quad t = D\ n_2 \quad D^k n_2 \leq \Gamma(x)}{\Gamma;\Delta \vdash x := (t)y \mid \Delta \cup \{x\};\emptyset}\ (\textsc{TCast})$$

$$\frac{\Gamma;\Delta \vdash s_1 \mid \Delta_1;\Sigma_1 \quad \Gamma;\Delta_1 \vdash s_2 \mid \Delta_2;\Sigma_2}{\Gamma;\Delta \vdash s_1; s_2 \mid \Delta_2;\Sigma_1 \cup \Sigma_2}\ (\textsc{TSeq})$$

Fig. 4. Statement typing.

constructor is also well-formed ($\vdash_C C$). These judgements are defined in Figure 5.

$$\frac{\mathit{mBody}(C,m) = s \quad \mathit{mSig}(C,m) = (\gamma, \overrightarrow{x_i{:}S_i}, S, \overrightarrow{y_j{:}S_j}) \quad \{\overrightarrow{\theta_l}\} = \mathit{instances}(C^\gamma!, \overrightarrow{S_i}) \quad \neg \mathit{nullable}(S) \Rightarrow \mathsf{res} \in \Delta \quad \overrightarrow{(x_i{:}\theta_l(S_i)}, \mathsf{this}{:}C^{\theta_l(\gamma)}!, \overrightarrow{y_j{:}\theta_l(S_j)}, \mathsf{res}{:}\theta_l(S)); \{\overrightarrow{x_i}, \mathsf{this}\} \vdash s \mid \Delta; \Sigma}{\vdash_m C, m}\ (\textsc{wfMeth})$$

$$\frac{\mathit{cBody}(C) = s \quad \mathit{cSig}(C) = (\overrightarrow{x_i{:}S_i}, \overrightarrow{y_j{:}S_j}) \quad \{\overrightarrow{\theta_l}\} = \mathit{instances}(\overrightarrow{S_i}) \quad \{f \mid f \in \mathit{fields}(C) \wedge \neg \mathit{nullable}(\mathit{fType}(C,f))\} \subseteq \Sigma \quad \overrightarrow{(x_i{:}\theta_l(S_i)}, \mathsf{this}{:}C^0!, \overrightarrow{y_j{:}\theta_l(S_j)}); \{\overrightarrow{x_i}, \mathsf{this}\} \vdash s \mid \Delta; \Sigma}{\vdash_C C, m}\ (\textsc{wfCons})$$

Fig. 5. Well-formed methods and constructors

3.3 Semantics

We adopt a reasonably standard heap model on which to define our operational semantics. Note that the heap model does not contain any type-system-specific information.

Definition 8 (Heaps, Values and Allocation). *We assume a finite set of* addresses*, ranged over by* ι*.*
Values*, ranged over by* v *are defined*[3] *by* $v ::= \iota \mid null$*.*
A heap h *is a pair* (h_v, h_c) *of partial functions;* h_v *from pairs of address and field-name to values, and* h_c *from addresses to class names. The domains of the functions are related by:* $dom(h_c) = \{\iota \mid \exists f.(\iota, f) \in dom(h_v)\}$*. As shorthand, we will typically use* h *in place of* h_v *or* h_c*.*
We write heap lookup *as* $h(\iota, f)$ *(defined as* $h_v(\iota, f)$*, only when* $(\iota, f) \in dom(h_v)$*).*
We write $h[(\iota, f) \mapsto v]$ *for* heap update *(meaning standard map update of* h_v*).*
We write class lookup *as* $cls(h, \iota)$*, meaning* $h_c(\iota)$ *(provided that* $\iota \in dom(h_c)$*).*

We model object allocation via a function alloc which takes a heap and a class-name as parameters, and returns a pair of heap and address, satisfying the following properties:

$$(h', \iota) = alloc(h, C) \Rightarrow \begin{cases} \iota \notin dom(h_c) \\ h'_v = h_v[\overrightarrow{(\iota, f_i) \mapsto null}] \quad where\ \overrightarrow{f_i} = fields(C) \\ h'_c = h_c[\iota \mapsto C] \end{cases}$$

A heap h_2 *is a* successor heap *of* h_1*, written* $h_1 \leq h_2$*, if* $dom(h_1) \subseteq dom(h_2)$ *and* $\forall \iota \in dom(h_1).\ cls(h_2, \iota) = cls(h_1, \iota)$*.*

We can now define the evaluation of expressions. Note that evaluation is not guaranteed per se to produce a value, since we might dereference a null variable. We model this by introducing an exception state (later, our main theorem will show that for a well-typed program, this exception state is never encountered).

Definition 9 (Expression evaluation). *A* stack frame σ *is a partial function from program variables to values. We write* $\sigma(x)$ *to denote the corresponding lookup (defined only when* $x \in dom(\sigma)$*), and we write* $\sigma[x \mapsto v]$ *for* stack update*.*
Extended Values*, ranged over by* V*, are either values* v *or the special symbol* derefExc *(denoting failure to obtain a value).*
Expression evaluation *maps an expression* e*, heap* h *and stack frame* σ *to an extended value. It is written* $\lfloor e \rfloor_{h,\sigma}$*, and defined as follows:*

$$\begin{aligned} \lfloor x \rfloor_{h,\sigma} &= \sigma(x) \\ \lfloor \mathsf{null} \rfloor_{h,\sigma} &= null \\ \lfloor x.f \rfloor_{h,\sigma} &= \begin{cases} h(\iota, f) & \text{if } \sigma(x) = \iota \text{ and } f \in fields(cls(h, \iota)) \\ \mathsf{derefExc} & \text{otherwise} \end{cases} \end{aligned}$$

We can now define the operational semantics of our language.

[3] Note that we use both null as an expression in the source language, and *null* as a distinguished value. However, the two are always distinguishable by context.

Definition 10 (Operational Semantics). Exception States*, ranged over by* ϵ*, are one of three possible concrete values:* $\epsilon ::= \mathsf{ok} \mid \mathsf{derefExc} \mid \mathsf{castExc}$.
Runtime Type Assignment *assigns simple types to runtime values, according to the subclassing relationship in the program. It is defined in Fig. 6. We define a big-step operational semantics via judgements* $\epsilon, h, \sigma, s \rightsquigarrow h', \sigma', \epsilon'$*, indicating the execution of statement* s *starting in exception state* ϵ*, heap* h *and stack-frame* σ*, and finishing with heap* h'*, stack-frame* σ *and exception state* ϵ'*. The rules are defined in Fig. 7.*

$$\frac{}{h \vdash \mathit{null} : C?}\ (\textsc{RNull}) \qquad \frac{\mathit{cls}(h, \iota) \leq C}{h \vdash \iota : C\ n}\ (\textsc{RAddr})$$

Fig. 6. Runtime type assignment.

3.4 Soundness Results

We can now turn to the formalisation of our soundness results. Firstly, we need to formally define our initialisation and reachability concepts.

Definition 11 (Initialisation and Reachability). *An address is* locally initialised *in a heap, written* $\mathit{init}(h, \iota)$*, if all non-null fields contain non-null values:*

$$\mathit{init}(h, \iota) \Leftrightarrow (\forall f \in \mathit{fields}(\mathit{cls}(h, \iota)) : \neg \mathit{nullable}(\mathit{fType}(\mathit{cls}(h, \iota), f)) \Rightarrow h(\iota, f) \neq \mathit{null})$$

An address reaches *another address in a heap, written* $\mathit{reaches}(h, \iota_1, \iota_2)$*, as defined recursively by the following:*

$$\mathit{reaches}(h, \iota_1, \iota_2) \Leftrightarrow \iota_1 = \iota_2 \vee \exists f, \iota_3 : h(\iota_1, f) = \iota_3 \wedge \mathit{reaches}(h, \iota_3, \iota_2)$$

Given an address and heap, the set of addresses reachable*, written* $\mathit{reachable}(h, \iota)$ *is defined by:* $\mathit{reachable}(h, \iota) = \{\iota' \mid \mathit{reaches}(h, \iota, \iota')\}$*. For convenience, we extend this concept to values by defining* $\mathit{reachable}(h, \mathit{null}) = \emptyset$.
An address is deeply initialised *in a heap, written* $\mathit{deep\_init}(h, \iota)$*, if all reachable addresses are locally initialised:*

$$\mathit{deep\_init}(h, \iota) \Leftrightarrow \forall \iota' \in \mathit{reachable}(h, \iota) : \mathit{init}(h, \iota')$$

Now, we are in a position to specify exactly what our type system preserves about the stack and the heap. We identify five conditions which go together to make up a "good" configuration. The first just forces the stack to have a suitable domain, while the second is the standard property that fields contain only objects which agree with their declared class type. The third expresses the meaning of our definite assignment checks for local variables, and the fourth expresses that stack variables which have been initialised contain suitable values. Finally, we characterise the type invariants of our system: committed references are deeply initialised and cannot reach objects directly referred to by free references.

$$\frac{\lfloor e \rfloor_{h,\sigma} = v}{\mathsf{ok}, h, \sigma, x := e \ \rightsquigarrow \ h, \sigma[x \mapsto v], \mathsf{ok}} \ (\textsc{varAss})$$

$$\frac{\lfloor e \rfloor_{h,\sigma} = \mathsf{derefExc}}{\mathsf{ok}, h, \sigma, x := e \ \rightsquigarrow \ h, \sigma, \mathsf{derefExc}} \ (\textsc{varAssBad})$$

$$\frac{\sigma(x) = \iota}{\mathsf{ok}, h, \sigma, x.f := y \ \rightsquigarrow \ h[(\iota, f) \mapsto \sigma(y)], \sigma, \mathsf{ok}} \ (\textsc{fldAss})$$

$$\frac{\sigma(x) = null}{\mathsf{ok}, h, \sigma, x.f := y \ \rightsquigarrow \ h, \sigma, \mathsf{derefExc}} \ (\textsc{fldAssBad})$$

$$\frac{\begin{array}{c}\sigma(y) = \iota \quad C = cls(h, \iota) \quad mSig(C, m) = (\gamma, \overrightarrow{x_i{:}S_i}, S, \overrightarrow{y_j{:}S_j}) \\ \sigma_1 = \mathsf{this} \mapsto \iota, x_i \mapsto \sigma(z_i)\,, \mathsf{res} \mapsto null, y_j \mapsto null \\ mBody(C, m) = s \quad \mathsf{ok}, h, \sigma_1, s \ \rightsquigarrow \ h', \sigma', \epsilon\end{array}}{\mathsf{ok}, h, \sigma, x := y.m(\overrightarrow{z_i}) \ \rightsquigarrow \ h', \sigma[x \mapsto \sigma'(\mathsf{res})], \epsilon} \ (\textsc{call})$$

$$\frac{\sigma(y) = null}{\mathsf{ok}, h, \sigma, x := y.m(\overrightarrow{z_i}) \ \rightsquigarrow \ h, \sigma, \mathsf{derefExc}} \ (\textsc{callBad})$$

$$\frac{\begin{array}{c}cSig(C, m) = (\overrightarrow{x_i{:}S_i}, \overrightarrow{y_j{:}S_j}) \quad (h_1, \iota_1) = alloc(h, C) \\ \sigma_1 = \mathsf{this} \mapsto \iota_1, x_i \mapsto \sigma(z_i)\,, y_j \mapsto null \quad cBody(C) = s \\ \mathsf{ok}, h_1, \sigma_1, s \ \rightsquigarrow \ h', \sigma_2, \epsilon\end{array}}{\mathsf{ok}, h, \sigma, x := \mathsf{new}\ C(\overrightarrow{z_i}) \ \rightsquigarrow \ h', \sigma[x \mapsto \iota_1], \epsilon} \ (\textsc{create})$$

$$\frac{h \vdash \sigma(y) : t}{\mathsf{ok}, h, \sigma, x := (t)y \ \rightsquigarrow \ h, \sigma[x \mapsto \sigma(y)], \mathsf{ok}} \ (\textsc{cast})$$

$$\frac{h \nvdash \sigma(y) : t}{\mathsf{ok}, h, \sigma, x := (t)y \ \rightsquigarrow \ h, \sigma, \mathsf{castExc}} \ (\textsc{castBad})$$

$$\frac{\mathsf{ok}, h, \sigma, s_1 \ \rightsquigarrow \ h_1, \sigma_1, \mathsf{ok} \quad \mathsf{ok}, h_1, \sigma_1, s_2 \ \rightsquigarrow \ h_2, \sigma_2, \epsilon}{\mathsf{ok}, h, \sigma, s_1; s_2 \ \rightsquigarrow \ h_2, \sigma_2, \epsilon} \ (\textsc{seq})$$

$$\frac{\mathsf{ok}, h, \sigma, s_1 \ \rightsquigarrow \ h_1, \sigma_1, \epsilon \quad \epsilon \neq \mathsf{ok}}{\mathsf{ok}, h, \sigma, s_1; s_2 \ \rightsquigarrow \ h_1, \sigma_1, \epsilon} \ (\textsc{seqBad})$$

Fig. 7. Operational semantics.

Definition 12 (Good Configurations). *A pair of heap and stack-frame is a* good configuration for Γ, Δ, *written* $\Gamma; \Delta \vdash h, \sigma$, *if the following conditions hold:*

1. $dom(\sigma) = dom(\Gamma) \ \wedge \ \mathsf{this} \in dom(\sigma)$
2. $\forall \iota \in dom(h), f \in fields(cls(h, \iota)) : (h(\iota, f) \neq null \Rightarrow \ h(\iota, f) \in dom(h) \wedge cls(h, h(\iota, f)) \leq fType(cls(h, \iota), f))$
3. $\forall x \in dom(\sigma) : \ (\neg nullable(\Gamma(x)) \wedge x \in \Delta \Rightarrow \sigma(x) \neq null)$

4. $\forall x \in dom(\sigma) : \ (\sigma(x) \neq null \ \wedge \ \Gamma(x) = C^k n \Rightarrow h \vdash \sigma(x) : C \ n)$
5. $\forall x, y \in dom(\sigma) : \ (committed(\Gamma(x)) \Rightarrow deep\_init(h, \sigma(x)) \wedge (free(\Gamma(y)) \Rightarrow \neg reaches(h, \sigma(x), \sigma(y))))$

Armed with this definition, we can state our desired soundness theorem:

Theorem 1 (Preservation and Safety). *If* $\Gamma; \Delta \vdash h, \sigma$ *and* $\Gamma; \Delta \vdash s \mid \Delta'; \Sigma$ *and* $\mathsf{ok}, h, \sigma, s \ \rightsquigarrow \ h', \sigma', \epsilon$ *and* $\epsilon \neq \mathsf{castExc}$ *all hold, then* $\Gamma; \Delta' \vdash h', \sigma' \ \wedge \ \epsilon = \mathsf{ok}$.

The proof of this theorem is challenging for a number of reasons. Not only is the design of our approach centred around reachability in the heap, but we present "good configurations" as a property local to each particular stack-frame. This means that there is much work to do in the proof when we change stack frame, particularly for a method or constructor return. In fact, we identified a number of interesting properties of our formalisation (some of which were not initially obvious) which lead to the proof. For any well-typed statement execution in our semantics the following properties hold in addition to the properties claimed in the theorem:

1. The domain of the stack is preserved, and the domain of the heap only grows.
2. After execution of the statement, all non-null fields in Σ of the receiver object contain non-null values.
3. Non-null fields which were initialised before execution of the statement, are still initialised afterwards.
4. Objects locally initialised before the execution of the statement are still locally initialised afterwards.
5. Any objects newly-allocated during the execution of the statement are locally initialised afterwards.
6. Any object which is not locally initialised and reachable from a stack variable after execution,is reachable from a stack variable before execution.
7. If, after execution, an object ι is *reachable* from a *committed* stack variable, and ι as well as the object referred to by the stack variable exist before execution, then ι is reachable from a committed stack variable before execution.
8. If, after execution, an object ι_1 *reaches* an object ι_2 referred to by a *free* stack variable, and both objects exist before execution, then ι_1 reaches an object referred to by a free stack variable before execution.
9. If an object ι_1 reaches another ι_2 after execution, and both objects exist before execution, then at least one of the following properties must hold before execution: (a) ι_1 reaches ι_2. (b) ι_2 can be reached from a committed stack variable. (c) ι_1 reaches an object referred to by a free stack variable, and ι_2 can be reached from a (possibly different) stack variable.

Property 9 deserves explanation. It essentially reflects the connecting of objects that can possibly happen during execution. Because committed references can be assigned to any fields, an object reachable from a committed local variable before execution could potentially be reachable by any object after execution. The only other kind of field assignment we allow, is the assignment of references to the fields of *free* references. In this case, an object which newly reaches another

must have previously reached the receiver of such a field update, that is, a free reference. We use all of the above-mentioned properties to strengthen our induction hypothesis when proving our main theorem; we prove the following lemma, which includes properties 1–9 as additional conclusions:

Lemma 1 (Preservation and Safety (strengthened)). *If* $\Gamma; \Delta \vdash h, \sigma$ *and* $\Gamma; \Delta \vdash s \mid \Delta'; \Sigma$ *and* $\mathsf{ok}, h, \sigma, s \rightsquigarrow h', \sigma', \epsilon$ *and* $\epsilon \neq \mathsf{castExc}$ *all hold, then:*

0. $\Gamma; \Delta' \vdash h', \sigma' \ \wedge \ \epsilon = \mathsf{ok}$
1. $\sigma'(\mathsf{this}) = \sigma(\mathsf{this}) \ \wedge \ dom(\sigma') = dom(\sigma) \ \wedge \ h \leq h'$
2. $\forall f \in \Sigma : \ (\neg nullable(fType(cls(h, \sigma(\mathsf{this})), f)) \Rightarrow h'(\sigma(\mathsf{this}), f) \neq null)$
3. $\forall \iota \in dom(h) : (\neg nullable(fType(cls(h, \iota), f)) \wedge h(\iota, f) \neq null \Rightarrow h'(\iota, f) \neq null)$
4. $\forall \iota \in dom(h) : \ (init(h, \iota) \Rightarrow init(h', \iota))$
5. $\forall \iota \in dom(h') : \ (\iota \notin dom(h) \Rightarrow init(h', \iota))$
6. $\forall \iota \in dom(h'), x \in dom(\sigma') : \ (reaches(h', \sigma'(x), \iota) \wedge \neg init(h', \iota) \Rightarrow (\exists y \in dom(\sigma) : \ reaches(h, \sigma(y), \iota)))$
7. $\forall \iota \in dom(h'), x \in dom(\sigma') : \ (reaches(h', \sigma'(x), \iota) \wedge committed(\Gamma(x)) \wedge \iota \in dom(h) \wedge \sigma'(x) \in dom(h) \Rightarrow (\exists y \in dom(\sigma) : \ committed(\Gamma(y)) \wedge reaches(h, \sigma(y), \iota)))$
8. $\forall \iota \in dom(h'), x \in dom(\sigma') : \ (reaches(h', \iota, \sigma'(x)) \wedge free(\Gamma(x)) \wedge \iota \in dom(h) \wedge \sigma'(x) \in dom(h) \Rightarrow (\exists y \in dom(\sigma) : \ free(\Gamma(y)) \wedge reaches(h, \iota, \sigma(y))))$
9. $\forall \iota_1 \in dom(h), \iota_2 \in dom(h) : \ (reaches(h', \iota_1, \iota_2) \Rightarrow reaches(h, \iota_1, \iota_2) \vee (\exists x \in dom(\sigma) : \ committed(\Gamma(x)) \wedge reaches(h, \sigma(x), \iota_2)) \vee (\exists y \in dom(\sigma), z \in dom(\sigma) : free(\Gamma(y)) \wedge free(\Gamma(z)) \wedge reaches(h, \iota_1, \sigma(y)) \wedge reaches(h, \sigma(z), \iota_2)))$

Proof. At the end of this section.

We need a number of auxiliary lemmas for the main proof.

Lemma 2 (Generation Lemmas).

1. $\Gamma; \Delta \vdash e : T \wedge T \leq C^0 n \Rightarrow free(T) \wedge (e = null \vee \exists x.(e = x \wedge free(\Gamma(x))))$
2. $\Gamma; \Delta \vdash e : T \wedge T \leq C^1 n \Rightarrow committed(T) \wedge (e = null \vee \exists x.((e = x \vee e = x.f) \wedge committed(\Gamma(x))))$
3. $\Gamma; \Delta \vdash e : T \wedge T \leq C^k! \Rightarrow \neg nullable(T) \wedge \exists x.((e = x \vee e = x.f) \wedge committed(\Gamma(x)))$
4. $\Gamma; \Delta \vdash y : T \Rightarrow T = \Gamma(y) \wedge y \in \Delta$
5. $\Gamma; \Delta \vdash y.f : T \Rightarrow y \in dom(\Gamma)$

Proof. All straightforward from the definitions.

Lemma 3 (Signature Type Instantiations).

1. *If* $S \leq S'$ *and* $dom(\theta) \supseteq vars(S, S')$ *then* $\theta(S) \leq \theta(S')$.
2. *If* $mSig(D, m) = (\gamma, \overrightarrow{x_i{:}S_i}, S, \overrightarrow{y_j{:}S_j})$ *and* $instance(\theta, \{D^\gamma!, \overrightarrow{S_i}\})$ *then all of the following are types (rather than signature types):* $D^{\theta(\gamma)}!, \overrightarrow{\theta(S_i)}, \theta(S), \overrightarrow{\theta(S_j)}$.
3. *If* $instance(\theta, \{\overrightarrow{S_i}\})$ *holds, then there exists* $\theta' \in instances(\overrightarrow{S_i})$ *such that* $\theta'(S_i) = \theta(S_i)$.

4. *If* $D \leq C$ *and we have* $mSig(D, m) = (\gamma, \overrightarrow{x_i{:}S_i}, S, \overrightarrow{y_j{:}S_j})$ *and* $mSig(C, m) = (\gamma', \overrightarrow{x_i{:}S'_i}, S', \overrightarrow{y_l{:}S_l})$ *and* $instance(\theta, \{D^{\gamma}!, \overrightarrow{S_i}\})$ *then, there exists* θ' *such that* $\theta' \in instances(C^{\gamma'}!, \overrightarrow{S'_i})$ *and* $\theta'(\gamma') = \theta(\gamma')$ *and* $\overrightarrow{\theta'(S'_i) = \theta(S'_i)}$ *and* $\theta'(S') = \theta(S)$ *and* $\overrightarrow{\theta'(S_l) = \theta(S_l)}$.

Proof. 1. *Immediate from Definition 2.*
2. *By the definition of instances (Definition 3) we know that all of the expectation variables in* γ *and* $\overrightarrow{S_i}$ *are in the domain of* θ*. By the restrictions imposed on method signatures in the same definition, the return types and local variable types may not mention any more than these expectation variables.*
3. *The substitution* θ' *can be obtained by taking the restriction (in the usual sense of function/map domain restriction) of* θ *to the* $vars(\overrightarrow{S_i})$*.*
4. *By the previous parts, it is enough to show that* $instance(\theta, \{C^{\gamma'}!, \overrightarrow{S'_i}\})$ *holds. By the restrictions on method signatures, we know that* $\gamma \leq \gamma'$ *and* $\overrightarrow{S_i \leq S'_i}$ *. By Definition 2, we observe that each supertype in these equations (the types on the right-hand side) cannot contain more expectation variables than the corresponding subtype. This gives us our result.*

Lemma 4 (Runtime Type Assignment Properties).

1. $h \vdash v : t_1 \wedge t_1 \leq t_2 \Rightarrow h \vdash v : t_2$
2. $h \vdash v : C\ n \Rightarrow v = null \vee cls(h, v) \leq C$
3. $\Gamma; \Delta \vdash h, \sigma\ \wedge\ \Gamma; \Delta \vdash e : C^k n \Rightarrow \exists v.(\lfloor e \rfloor_{h,\sigma} = v \wedge h \vdash v : C\ n)$

Proof. The first two parts require a straightforward case analysis on the definitions. The third is slightly more involved, but requires only a straightforward induction on the structure of e*, with a case-split on the initialisation expectation on* x *in the case of* $e = x.f$*.*

Lemma 5 (Heap Update Properties).

1. $\lfloor x.f \rfloor_{h,\sigma} = v \Rightarrow reachable(h, v) \subseteq reachable(h, \sigma(x))$
2. $h' = h[(\iota, f) \mapsto v] \wedge \iota \in reachable(h, \iota') \Rightarrow reachable(h', \iota') \subseteq reachable(h, v) \cup reachable(h, \iota')$
3. $h' = h[(\iota, f) \mapsto v] \wedge \iota \notin reachable(h, \iota') \Rightarrow reachable(h', \iota') = reachable(h, \iota')$
4. $(h', \iota') = alloc(h, C) \Rightarrow \forall \iota \in dom(h).(\neg reaches(h', \iota, \iota') \wedge \neg reaches(h', \iota', \iota))$

Proof. The first part is straightforward from the definitions. The next two parts follow by induction on the derivation of $reachable(h, \iota')$*. The last follows from Definition 8.*

We can now prove our main result.

Proof (of Lemma 1). For reference, we enumerate the assumptions of our lemma (including expansion of Definition 12):

$$\mathsf{ok}, h, \sigma, s \ \rightsquigarrow\ h', \sigma', \epsilon \tag{1}$$

$$\epsilon \neq \mathsf{castExc} \tag{2}$$

$$dom(\sigma) = dom(\Gamma) \ \wedge \ \mathsf{this} \in dom(\sigma) \tag{3}$$

$$\begin{array}{c}\forall \iota \in dom(h), f \in \mathit{fields}(\mathit{cls}(h,\iota)) : \\ (h(\iota,f) \neq \mathit{null} \Rightarrow h(\iota,f) \in dom(h) \wedge \mathit{cls}(h,h(\iota,f)) \leq \mathit{fType}(\mathit{cls}(h,\iota),f))\end{array} \tag{4}$$

$$\forall x \in dom(\sigma) : \ (\neg \mathit{nullable}(\Gamma(x)) \wedge x \in \Delta \Rightarrow \sigma(x) \neq \mathit{null}) \tag{5}$$

$$\forall x \in dom(\sigma) : \ (\sigma(x) \neq \mathit{null} \Rightarrow h \vdash \sigma(x) : \Gamma(x)) \tag{6}$$

$$\begin{array}{c}\forall x, y \in dom(\sigma) : \ (\mathit{committed}(\Gamma(x)) \Rightarrow \mathit{deep\_init}(h,\sigma(x)) \\ \wedge (\mathit{free}(\Gamma(y)) \Rightarrow \neg \mathit{reaches}(h,\sigma(x),\sigma(y))))\end{array} \tag{7}$$

$$\Gamma; \Delta \vdash s \mid \Delta'; \Sigma \tag{8}$$

We proceed by induction on the derivation of (*Eq.* 1), considering cases for the last rule applied. In each case, we have to prove all of the following:

$$dom(\sigma') = dom(\Gamma) \ \wedge \ \mathsf{this} \in dom(\sigma') \tag{9}$$

$$\begin{array}{c}\forall \iota \in dom(h'), f \in \mathit{fields}(\mathit{cls}(h',\iota)) : \\ (h'(\iota,f) \neq \mathit{null} \Rightarrow h'(\iota,f) \in dom(h') \wedge \mathit{cls}(h',h'(\iota,f)) \leq \mathit{fType}(\mathit{cls}(h',\iota),f))\end{array} \tag{10}$$

$$\forall x \in dom(\sigma') : \ (\neg \mathit{nullable}(\Gamma(x)) \wedge x \in \Delta' \Rightarrow \sigma'(x) \neq \mathit{null}) \tag{11}$$

$$\forall x \in dom(\sigma') : \ (\sigma'(x) \neq \mathit{null} \Rightarrow h' \vdash \sigma'(x) : \Gamma(x)) \tag{12}$$

$$\begin{array}{c}\forall x, y \in dom(\sigma') : \ (\mathit{committed}(\Gamma(x)) \Rightarrow \mathit{deep\_init}(h',\sigma'(x)) \\ \wedge (\mathit{free}(\Gamma(y)) \Rightarrow \neg \mathit{reaches}(h',\sigma'(x),\sigma'(y))))\end{array} \tag{13}$$

$$\epsilon = \mathsf{ok} \tag{14}$$

$$\sigma'(\mathsf{this}) = \sigma(\mathsf{this}) \ \wedge \ dom(\sigma') = dom(\sigma) \ \wedge \ h \leq h' \tag{15}$$

$$\forall f \in \Sigma : \ (\neg \mathit{nullable}(\mathit{fType}(\mathit{cls}(h,\sigma(\mathsf{this})),f)) \Rightarrow h'(\sigma(\mathsf{this}),f) \neq \mathit{null}) \tag{16}$$

$$\forall \iota \in dom(h) : (\neg \mathit{nullable}(\mathit{fType}(\mathit{cls}(h,\iota),f)) \wedge h(\iota,f) \neq \mathit{null} \Rightarrow h'(\iota,f) \neq \mathit{null}) \tag{17}$$

$$\forall \iota \in dom(h') : \ (\iota \notin dom(h) \Rightarrow \mathit{init}(h',\iota)) \tag{18}$$

$$\begin{array}{c}\forall \iota \in dom(h'), x \in dom(\sigma') : \ (\mathit{reaches}(h',\sigma'(x),\iota) \wedge \iota \in dom(h) \\ \Rightarrow (\exists y \in dom(\sigma) : \ \mathit{reaches}(h,\sigma(y),\iota)))\end{array} \tag{19}$$

$$\begin{array}{c}\forall \iota \in dom(h'), x \in dom(\sigma') : \ (\mathit{reaches}(h',\sigma'(x),\iota) \wedge \mathit{committed}(\Gamma(x)) \\ \wedge \iota \in dom(h) \wedge \sigma'(x) \in dom(h) \Rightarrow \\ (\exists y \in dom(\sigma) : \ \mathit{committed}(\Gamma(y)) \wedge \mathit{reaches}(h,\sigma(y),\iota)))\end{array} \tag{20}$$

$$\begin{array}{c}\forall \iota \in dom(h'), x \in dom(\sigma') : \ (\mathit{reaches}(h',\iota,\sigma'(x)) \wedge \mathit{free}(\Gamma(x)) \\ \wedge \iota \in dom(h) \wedge \sigma'(x) \in dom(h) \\ \Rightarrow (\exists y \in dom(\sigma) : \ \mathit{free}(\Gamma(y)) \wedge \mathit{reaches}(h,\iota,\sigma(y))))\end{array} \tag{21}$$

$$\begin{array}{c}\forall \iota_1 \in dom(h), \iota_2 \in dom(h) : \ (\mathit{reaches}(h',\iota_1,\iota_2) \Rightarrow \\ \mathit{reaches}(h,\iota_1,\iota_2) \vee \\ (\exists x \in dom(\sigma) : \ \mathit{committed}(\Gamma(x)) \wedge \mathit{reaches}(h,\sigma(x),\iota_2)) \vee \\ (\exists y \in dom(\sigma), z \in dom(\sigma) : \ \mathit{free}(\Gamma(y)) \wedge \mathit{reaches}(h,\iota_1,\sigma(y)) \wedge \mathit{reaches}(h,\sigma(z),\iota_2)))\end{array} \tag{22}$$

Note that we omit the goal $\forall \iota \in dom(h) : \ (\mathit{init}(h,\iota) \Rightarrow \mathit{init}(h',\iota))$ since it follows straightforwardly from (**Goal 17**) above.

We now proceed with our case analysis:

(**varAss**) **:** Then, from the form of the rule, we have:

$$\epsilon = \mathsf{ok} \tag{23}$$
$$h = h' \tag{24}$$
$$\sigma' = \sigma[x \mapsto v] \tag{25}$$
$$s = (x := e) \tag{26}$$
$$\lfloor e \rfloor_{h,\sigma} = v \tag{27}$$

Note that (**Goal 14**) is given by (*Eq.* 23).
By (*Eq.* 26), (*Eq.* 8) must have been derived by rule (TVarAss), and so we also have:

$$\Delta' = \Delta \cup \{x\} \tag{28}$$
$$\Sigma = \emptyset \tag{29}$$
$$x \in dom(\Gamma) \tag{30}$$
$$\Gamma; \Delta \vdash e : T \tag{31}$$
$$T \leq \Gamma(x) \tag{32}$$

By (*Eq.* 25), using (*Eq.* 30) and (*Eq.* 3), we know that

$$dom(\sigma') = dom(\sigma) \cup \{x\} = dom(\sigma) = dom(\Gamma)$$

Therefore, we deduce (**Goal 9**) and (**Goal 15**) (Note that from the syntax of statements, we know that $x \neq \mathsf{this}$). From (*Eq.* 24) we also immediately deduce (**Goal 17**), (**Goal 19**), (**Goal 18**) and (**Goal 10**). From (*Eq.* 29) we deduce (**Goal 16**) vacuously.
To prove (**Goal 13**), we need only concern ourselves with the updated variable x (by (*Eq.* 6),(*Eq.* 24) and (*Eq.* 25)). The only interesting cases are when either $committed(\Gamma(x))$ or $free(\Gamma(x))$ hold. In the former case, by (*Eq.* 32) and Lemma 2(2) we know that either $e = null$ or, for some y with $committed(\Gamma(y))$ we have $e = y$ or $e = y.f$. Then $deep\_init(h', \sigma'(x))$ follows, using (*Eq.* 7) and Lemma 5(1) as needed if $e \neq null$. In the other case of $free(\Gamma(x))$ holding, we know by Lemma 2(1) that $e = null$ or $e = y$ with $free(\Gamma(y))$. The requirement that free references remain unreachable from committed references follows from (*Eq.* 7). Similarly to the above arguments, we can use Lemma 2(2) to obtain (**Goal 11**) (using (*Eq.* 28)) and (**Goal 20**), and we can use Lemma 2(1) to obtain (**Goal 21**).
We obtain (**Goal 12**) by Lemma 4(3) and using (*Eq.* 6), (*Eq.* 5), (*Eq.* 31) and (*Eq.* 27). Note that showing (**Goal 22**) is made trivial by (*Eq.* 24).

(**varAssBad**) **:** From the form of the rule, we know that $\lfloor e \rfloor_{h,\sigma} = \mathsf{derefExc}$. By Definition 9, we must have $e = x.f$ with either $\sigma(x) = null$ or $\sigma(x) = \iota$ with $f \notin fields(cls(h, \iota))$. Then (*Eq.* 8) must have been derived by rule (TVarAss). In particular, it must be the case that $\Gamma; \Delta \vdash x.f : T$. From Fig. 3 we see that this implies that, for some C and k we have $\Gamma; \Delta \vdash x : C^k!$ and $f \in fields(C)$. By Lemma 2(4) we have $\Gamma(x) = C^k!$ and $x \in \Delta$. By

(*Eq.* 5) we obtain $\sigma(x) \neq \mathit{null}$. Therefore, we must have $\sigma(x) = \iota$ with $f \notin \mathit{fields}(\mathit{cls}(h, \iota))$. But, by (*Eq.* 6) we can conclude that $h \vdash \iota : C^k!$, and thus, $\mathit{cls}(h, \iota) \leq C$ from Fig. 6. By Definition 1 we have $f \in \mathit{fields}(\mathit{cls}(h, \iota))$; a contradiction.

(**FldAss**) **:** From the form of the rule, we have:

$$\epsilon = \mathsf{ok} \tag{33}$$
$$s = (x.f := y) \tag{34}$$
$$h' = h[(\iota, f) \mapsto \sigma(y)] \tag{35}$$
$$\sigma' = \sigma \tag{36}$$
$$\sigma(x) = \iota \tag{37}$$

Note that (**Goal 14**) is given by (*Eq.* 23), and (**Goal 18**) follows easily from (*Eq.* 35) and (*Eq.* 4).

By (*Eq.* 34), we know that (*Eq.* 8) must have been derived by rule (TFldAss). Therefore, we also have

$$\Gamma; \Delta \vdash x : C^{k_1}! \tag{38}$$
$$\mathit{fType}(C, f) = D\ n \tag{39}$$
$$\Gamma; \Delta \vdash y : T \tag{40}$$
$$T \leq D^{k_2} n \tag{41}$$
$$k_1 = 0 \vee k_2 = 1 \tag{42}$$
$$\Sigma = \{f\}\ \mathit{if}\ x = \mathsf{this},\ \emptyset\ \mathit{otherwise} \tag{43}$$

By using (*Eq.* 3) and (*Eq.* 5) with (*Eq.* 36), we get (**Goal 9**) and (**Goal 11**). With (*Eq.* 35) we then obtain (**Goal 15**), which also gives us (**Goal 12**) (by $h \leq h'$). By Lemma 4(3) (using (*Eq.* 40)) and Lemma 4(1) (using (*Eq.* 41)) we obtain

$$h \vdash \sigma(y) : D\ n \tag{44}$$

Using this, along with Lemma 4(2) and (*Eq.* 35), we obtain (**Goal 10**).

To show (**Goal 16**), the only interesting case is when both $x = \mathsf{this}$ and $\neg \mathit{nullable}(\mathit{fType}(\mathit{cls}(h, \sigma(\mathsf{this}))))$ hold. From (*Eq.* 38) and Lemma 2(4) we have

$$C^{k_1}! = \Gamma(x) \tag{45}$$

and using (*Eq.* 6) we obtain $h \vdash \sigma(\mathsf{this}) : C\ !$. By Lemma 4(2), we have

$$\mathit{cls}(h, \sigma(\mathsf{this})) \leq C \tag{46}$$

By invariance of *fType*, we have (with (*Eq.* 39)) $\neq \mathit{nullable}(D\ n)$, i.e., $n =!$. By Lemma 2(3), $\neq \mathit{nullable}(T)$. Therefore, using (*Eq.* 44) we have $\sigma(y) \neq \mathit{null}$, from which we obtain (**Goal 16**). By similar argument, we obtain (**Goal 17**).

To obtain (**Goal 19**), let us suppose that for some z and ι' we have both $\mathit{reaches}(h', \sigma'(z), \iota')$ and $\iota' \in \mathit{dom}(h)$. Note that by (*Eq.* 36) we have $\sigma(z) = \sigma'(z)$. We need to show that $\exists w.(\mathit{reaches}(h, \sigma(w), \iota'))$. We consider two cases (depending on whether the variable in question "saw" the field update):

reaches$(h, \sigma(z), \iota)$ **:** By applying Lemma 5(2) we can deduce $\mathit{reachable}(h', \sigma'(z)) \subseteq \mathit{reachable}(h, \sigma(z)) \cup \mathit{reachable}(h, \sigma(y))$. Therefore, $\mathit{reaches}(h', \sigma'(z), \iota')$ implies that either $\mathit{reaches}(h, \sigma(z), \iota')$ (in which case we are done, choosing w to be z), or $\mathit{reaches}(h, \sigma(y), \iota')$ (in which we are done, choosing w to be y).

$\neg$***reaches***$(h, \sigma(z), \iota)$ **:** By Lemma 5(3) we have $\mathit{reachable}(h', \sigma'(z)) = \mathit{reachable}(h, \sigma(z))$, and we are done, picking w to be z.

We still need to show the following: (**Goal 13**),(**Goal 21**),(**Goal 20**) and (**Goal 22**). We show all of these by case analysis on (*Eq.* 42):

$(k_1 = 0)$ **:** By (*Eq.* 45), we have $\mathit{free}(\Gamma(x))$.

Let w be an arbitrary variable such that $\mathit{committed}(\Gamma(w))$ holds. By (*Eq.* 7) we know that $\neg\mathit{reaches}(h, \sigma(w), \sigma(x)))$. By Lemma 5(3) we know that $\mathit{reachable}(h', \sigma'(w)) = \mathit{reachable}(h, \sigma(y))$, from which (**Goal 13**) follows.

To show (**Goal 21**) we consider two cases (for arbitrary ι'):

$(\sigma(x) \in$ ***reachable***$(h, \iota'))$ **:** Then we are done, since $\mathit{free}(\Gamma(x))$ holds.

$(\sigma(x) \notin$ ***reachable***$(h, \iota'))$ **:** Then, by Lemma 5(3), $\mathit{reachable}(h', \iota') = \mathit{reachable}(h, \iota')$, from which the implication easily follows.

We can show (**Goal 22**) by similar argument, considering (for arbitrary ι') cases for whether or not $\mathit{reachable}(h, \iota', \sigma(x))$ holds.

To show (**Goal 20**), suppose that for some z and $\iota' \in \mathit{dom}(h)$ we have $\mathit{reachable}(h', \sigma'(z), \iota') \wedge \mathit{committed}(\Gamma(z))$. By (*Eq.* 7) we know that $\neg\mathit{reaches}(h, \sigma(z), \sigma(x))$. By Lemma 5(3) we conclude $\mathit{reachable}(h, \sigma(z), \iota')$.

$(k_2 = 1)$ **:** To show (**Goal 13**), let z be an arbitrary variable such that $\mathit{committed}(\Gamma(z))$ holds. We consider two cases:

$($***reaches***$(h, \sigma(z), \sigma(x)))$ **:** Then we conclude by Lemma 5(3) as above.

$(\neg$***reaches***$(h, \sigma(z), \sigma(x)))$ **:** Then by Lemma 5(2), $\mathit{reachable}(h', \sigma'(z)) \subseteq \mathit{reachable}(h, \sigma(z)) \cup \mathit{reachable}(h, \sigma(y))$. By (*Eq.* 7) and (**Goal 17**), it is enough to show that $\forall \iota' \in \mathit{reachable}(h, \sigma(y)), \mathit{init}(h, \iota')$. By Lemma 2(2) and (*Eq.* 40) we have $\mathit{committed}(\Gamma(y))$, from which the required property follows by (*Eq.* 6).

To show (**Goal 21**) we consider two cases (for arbitrary ι'):

$(\sigma(x) \in$ ***reachable***$(h, \iota'))$ **:** By applying Lemma 5(2), we obtain $\mathit{reachable}(h', \iota') = \mathit{reachable}(h, \iota') \cup \mathit{reachable}(h, \sigma(y))$. However, by (*Eq.* 40) and (*Eq.* 41) and using Lemmas 2(2) and 2(4), we have $\mathit{committed}(\Gamma(y))$. Therefore, by (*Eq.* 7), we know that no variable of free type is reachable from $\sigma(y)$, which is enough to conclude the result.

$(\sigma(x) \notin$ ***reachable***$(h, \iota'))$ **:** Then, by Lemma 5(3), $\mathit{reachable}(h', \iota') = \mathit{reachable}(h, \iota')$, from which the implication easily follows.

The remaining (**Goal 20**) and (**Goal 22**) follow by similar arguments to those for the $k_1 = 0$ case.

(**FldAssBad**) **:** From the form of the rule, we have

$$s = x.f := y \tag{47}$$

$$\sigma(x) = \mathit{null} \tag{48}$$

By (*Eq.* 47) we know that (*Eq.* 8) must have been derived by rule (TFldAss). Therefore, we also know that

$$\Gamma; \Delta \vdash x : C^{k_1}! \tag{49}$$

By Lemma 2(4), we obtain $x \in \Delta$ and $\Gamma(x) = C^{k_1}!$. Therefore, by (*Eq.* 5), we have $\sigma(x) \neq \mathit{null}$ — a contradiction.

(**Call**) **:** From the form of the rule, we have:

$$s = (x := y.m(\overrightarrow{z_i})) \tag{50}$$
$$\sigma(y) = \iota \tag{51}$$
$$D = \mathit{cls}(h, \iota) \tag{52}$$
$$\mathit{mSig}(D, m) = (\gamma, \overrightarrow{x_i{:}S_i}, S, \overrightarrow{y_j{:}S_j}) \tag{53}$$
$$\sigma_1 = \mathsf{this}{\mapsto}\iota, \overrightarrow{x_i{\mapsto}\sigma(z_i)}, \mathsf{res}{\mapsto}\mathit{null}, \overrightarrow{y_j{\mapsto}\mathit{null}} \tag{54}$$
$$\mathit{mBody}(D, m) = s' \tag{55}$$
$$\mathsf{ok}, h, \sigma_1, s' \rightsquigarrow h', \sigma_2, \epsilon \tag{56}$$
$$\sigma' = \sigma[x{\mapsto}\sigma_2(\mathsf{res})] \tag{57}$$

From the rule (TCall) we also obtain:

$$\Delta' = \Delta \cup \{x\} \tag{58}$$
$$\Sigma = \emptyset \tag{59}$$
$$\Gamma; \Delta \vdash y : C^k! \tag{60}$$
$$\overrightarrow{\Gamma; \Delta \vdash z_i : T_i} \tag{61}$$
$$\mathit{mSig}(C, m) = (\gamma', \overrightarrow{x_i{:}S'_i}, S', \overrightarrow{y_l{:}S'_l}) \tag{62}$$
$$\mathit{instance}(\theta, \{\overrightarrow{S'_i}, S', C^{\gamma'}!\}) \tag{63}$$
$$\overrightarrow{T_i \leq \theta(S'_i)} \tag{64}$$
$$\theta(S') \leq \Gamma(x) \tag{65}$$
$$k \leq \theta(\gamma') \tag{66}$$

From $\Gamma; \Delta \vdash h, \sigma$ and (*Eq.* 51) we obtain $h \vdash \iota : \Gamma(y)$ and so by Lemma 4(2) we obtain $\mathit{cls}(h, \iota) \leq \Gamma(y)$. By the variance restrictions on method signatures (Definition 3), we obtain $\overrightarrow{S'_i \leq S_i}$ and $\gamma' \leq \gamma$ and $\overrightarrow{S_j \leq S'_j}$. Using these facts along with Lemma 3(1), we obtain:

$$\overrightarrow{T_i \leq \theta(S_i)} \tag{67}$$
$$\theta(S) \leq \Gamma(x) \tag{68}$$
$$k \leq \theta(\gamma) \tag{69}$$

By the method checks of Definition 7, and using Lemma 3(4), we obtain (defining Γ' and Δ' to allow conveniently referring to them later on):

$$\Gamma' = (\overrightarrow{x_i{:}\theta(S_i)}, \mathsf{this}{:}C^{\theta(\gamma)}, \mathsf{res}{:}\theta(S), \overrightarrow{y_j{:}\theta(S_j)}) \tag{70}$$
$$\Delta' = \{\overrightarrow{x_i}, \mathsf{this}\} \tag{71}$$
$$\Gamma'; \Delta' \vdash s' \mid \Delta''; \Sigma \tag{72}$$
$$\mathit{committed}(\theta(S)) \Rightarrow \mathsf{res} \in \Delta'' \tag{73}$$

We now aim to show $\Gamma'; \Delta' \vdash h, \sigma_1$ holds, to allow us to apply our induction hypothesis. Inspecting Definition 12, we need to show points 1-5. Point 1 holds by definition of Γ' and Δ'. Using $\Gamma; \Delta \vdash h, \sigma$ along with (*Eq.* 61) and Lemma 4(3), we obtain $h \vdash \sigma(z_i) : T_i$. By (*Eq.* 67) and Lemma 4(1), we obtain $h \vdash \sigma(z_i) : \theta(S_i)$. This, along with the fact that (by definition) $h \vdash \iota : \mathit{cls}(h, \iota)$ holds, gives us point 2. To show point 3, we need only consider those variables v such that $\mathit{committed}(\Gamma'(v))$ holds. The point follows from the corresponding point for $\Gamma; \Delta \vdash h, \sigma$, using Lemma 2(2) and (*Eq.* 67). Points 4 and 5 can be similarly derived from the corresponding points for $\Gamma; \Delta \vdash h, \sigma$.

We can therefore apply our induction hypothesis, using (*Eq.* 56). This tells us that all of our goals 9 to 22 hold, but with σ_2 in place of σ' (we will not enumerate all statements here, but will hereafter write "by I.H." for properties which follow from the induction hypothesis). We must therefore justify that these goals can also be deduced to hold for $\sigma' = \sigma[x \mapsto \sigma_2(\mathsf{res})]$.

Directly from the induction hypothesis, we obtain the following: (**Goal 9**), (**Goal 10**), (**Goal 14**), (**Goal 15**), (**Goal 16**), (**Goal 17**). For (**Goal 12**), we need to be sure that the return value $\sigma_2(\mathsf{res})$ satisfies $h' \vdash \sigma_2(\mathsf{res}) : \Gamma(x)$. By I.H. we know that $h' \vdash \sigma_2(\mathsf{res}) : \Gamma'(\mathsf{res})$ (since we have $\Gamma'; \Delta' \vdash h', \sigma_2$). We conclude by (*Eq.* 68) and Lemma 4(1). By assumption, $\epsilon \neq \mathsf{castExc}$, and from the I.H. we obtain (**Goal 14**).

(**Goal 11**) follows easily from the corresponding information in $\Gamma'; \Delta' \vdash h', \sigma_2$ - Lemma 2(3) tells us that if $\Gamma(x)$ is not nullable then $\Gamma'(\mathsf{res})$ also must not be (given (*Eq.* 68)).

For several of the remaining goals, the following observations (which will be referred to in the following proof) about the initialisation of σ_1 are helpful. Firstly, for any variable y (of the callee's stack frame) such that $\mathit{committed}(\Gamma'(y))$ holds, there exists a variable u (of the caller's stack frame) such that $\mathit{committed}(\Gamma(u))$ and $\sigma_1(y) = \sigma(u)$ hold (this relies on Lemma 2(2)). Similarly, for any variable y (of the callee's stack frame) such that $\mathit{free}(\Gamma'(y))$ holds, there exists a variable u (of the caller's stack frame) such that $\mathit{free}(\Gamma(u))$ and $\sigma_1(y) = \sigma(u)$ hold (this relies on Lemma 2(1)).

Using these observations, it is then easy to show (**Goal 22**) by case analysis on the corresponding property from the I.H.. (**Goal 18**) follows easily from the I.H. and (**Goal 22**). (**Goal 19**) follows from the I.H. and the observations above.

(**Goal 21**) requires a more elaborate argument. Suppose that, for some variable z we have $\mathit{reaches}(h', \iota, \sigma'(z))$ and $\mathit{free}(\Gamma(z))$ and $\iota \in \mathit{dom}(h)$ and $\sigma'(z) \in \mathit{dom}(h)$. We need to show that $\exists z'.\mathit{reaches}(h, \iota, \sigma(z')) \wedge \mathit{free}(\Gamma(z'))$ holds. We consider two cases:

($z \neq x$) **:** Then $\sigma'(z) = \sigma(z)$ holds. Using (**Goal 22**), we can consider three cases:

($\boldsymbol{\mathit{reaches}}(h, \iota, \sigma(z))$) **:** Then we are done, choosing z' to be z.

($\exists w.(\boldsymbol{\mathit{committed}}(\Gamma(w)) \wedge \boldsymbol{\mathit{reaches}}(h, \sigma_1(w), \sigma(z)))$) **:** Then, by our observations above, there must be a variable v such that $\mathit{committed}(\Gamma(v))$

and $\sigma(v) = \sigma_1(w)$ both hold. This leads to a contradiction with (*Eq.* 13).

($\exists w, v.(\textbf{\textit{free}}(\Gamma'(w)) \wedge \textbf{\textit{reaches}}(h, \iota, \sigma_1(w)) \wedge \textbf{\textit{reaches}}(h, \sigma_1(v), \sigma(z)))$ **:** Then, by the observations above, there exists a variable u with $\mathit{free}(\Gamma(u))$ and $\sigma(u) = \sigma_1(w)$. We are done, taking $z' = u$.

($z = x$) **:** Then $\sigma'(z) = \sigma_2(\mathsf{res})$ with $\mathit{free}(\Gamma'(\mathsf{res}))$. Then, using the version of (*Eq.* 21) from the induction hypothesis, we have that for some variable w, $\mathit{reaches}(h, \iota, \sigma_1(w)) \wedge \mathit{free}(\Gamma'(w))$ holds. By the observations above, there exists a variable u with $\mathit{free}(\Gamma(u))$ and $\sigma(u) = \sigma_1(w)$. We are done, taking $z' = u$.

We can show (**Goal 20**) and (**Goal 13**) by similar case analyses using (**Goal 22**) (and using (**Goal 18**) in the latter case, also). This completes the case.

(**CallBad**) **:** Leads to a contradiction, by analogous argument to the case for (FldAssBad).

(**Create**) **:** From the form of the rule, we have:

$$s = (x := \mathsf{new}\ C(\overrightarrow{z_i})) \tag{74}$$
$$\sigma' = \sigma[x \mapsto \iota_1] \tag{75}$$
$$(h_1, \iota_1) = \mathit{alloc}(h, C) \tag{76}$$
$$\mathit{cSig}(C) = (\overrightarrow{x_i{:}S_i}, \overrightarrow{y_j{:}S_j}) \tag{77}$$
$$\sigma_1 = \mathsf{this} \mapsto \iota_1, \overrightarrow{x_i \mapsto \sigma(z_i)}, \overrightarrow{y_j \mapsto \mathit{null}} \tag{78}$$
$$\mathit{cBody}(C) = s' \tag{79}$$
$$\mathsf{ok}, h_1, \sigma_1, s' \rightsquigarrow h', \sigma_2, \epsilon \tag{80}$$

From the rule (TCreate) we also have:

$$\mathit{cSig}(C) = (\overrightarrow{x_i{:}S_i}, \overrightarrow{y_j{:}S_j}) \tag{81}$$
$$\mathit{instance}(\sigma, \{\overrightarrow{S_i}\}) \tag{82}$$
$$\overrightarrow{\Gamma; \Delta \vdash z_i : T_i} \tag{83}$$
$$\overrightarrow{T_i \leq \sigma(S_i)} \tag{84}$$
$$k = \bigwedge \overrightarrow{\mathit{committed}(T_i)} \tag{85}$$
$$C^k! \leq \Gamma(x) \tag{86}$$
$$\Sigma = \emptyset \tag{87}$$
$$\Delta' = \Delta \cup \{x\} \tag{88}$$

By Lemma 3(3) and the method checks of Definition 7, we know (defining Γ' and Δ' to allow conveniently referring to them later on):

$$\Gamma' = (\overrightarrow{x_i{:}\sigma(S_i)}, \mathsf{this}{:}C^0!, \overrightarrow{y_j{:}\sigma(S_j)}) \tag{89}$$
$$\Delta' = \{\overrightarrow{x_i}, \mathsf{this}\} \tag{90}$$
$$\Gamma'; \Delta' \vdash s' \mid \Delta''; \Sigma' \tag{91}$$
$$\{f \mid f \in \mathit{fields}(C) \wedge \neg \mathit{nullable}(\mathit{fType}(C, f))\} \subseteq \Sigma' \tag{92}$$

We now aim to show $\Gamma'; \Delta' \vdash h_1, \sigma'$, so that we can apply induction. This requires us to show points 1-5 of Definition 12. Point 1 follows by construction, while point 2 follows by similar argument to that made for the method call case above. Point 3 follows from $\Gamma; \Delta \vdash h, \sigma$, using (*Eq.* 84), Lemma 2(1) and Lemma 2(2). Similarly, point 5 follows using Lemma 2(3). For point 4, we need only concern ourselves with the newly-allocated object at ι_1. However, this object trivially satisfies the requirements with its default initialisation (Definition 8).

We can therefore apply induction, using (*Eq.* 80). The following goals then follow straightforwardly, using the I.H.: (**Goal 9**), (**Goal 10**), (**Goal 11**), (**Goal 14**), (**Goal 15**), (**Goal 16**), (**Goal 17**).

(**Goal 12**) follows from (*Eq.* 75), the I.H. and (*Eq.* 86). (**Goal 18**) requires (given the I.H.) that we can justify that $\mathit{init}(h', \iota_1)$ holds. We get this from the version of (*Eq.* 16) in the I.H., along with (*Eq.* 92). (**Goal 22**) follows from the I.H. and the properties of *alloc* (Definition 8).

Similarly to the treatment of method calls, for several of the remaining goals, the following observations (which will be referred to in the following proof) about the initialisation of σ_1 are helpful. Firstly, for any variable y (of the callee's stack frame) such that $\mathit{committed}(\Gamma'(y))$ holds, there exists a variable u (of the caller's stack frame) such that $\mathit{committed}(\Gamma(u))$ and $\sigma_1(y) = \sigma(u)$ hold (this relies on Lemma 2(2)). Similarly, for any variable y (of the callee's stack frame) such that $\mathit{free}(\Gamma'(y))$ holds, there exists a variable u (of the caller's stack frame) such that $\mathit{free}(\Gamma(u))$ and $\sigma_1(y) = \sigma(u)$ hold (this relies on Lemma 2(1)).

For (**Goal 19**), we need to show that, for any variable w and address ι'' such that $\mathit{reaches}(h', \sigma'(w), \iota'') \wedge \iota'' \in \mathit{dom}(h)$ holds, we can deduce that $\exists u.(\mathit{reaches}(h, \sigma(u), \iota''))$. We consider two cases:

- $(w = x)$ **:** Using the corresponding property from the I.H., we obtain that $\exists v.(\mathit{reaches}(h, \sigma_1(v), \iota''))$ must hold. Since this variable was initialised to one of the parameters $\overrightarrow{z_i}$, we have the required result.
- $(w \neq x)$ **:** Then $\sigma(w) = \sigma'(w)$. We consider cases using the (already proven) (**Goal 22**):
  - $(\boldsymbol{\mathit{reaches}}(h, \sigma(w), \iota''))$ **:** Then we are done, choosing u to be w.
  - $(\exists v.(\boldsymbol{\mathit{committed}}(\Gamma(v)) \wedge \boldsymbol{\mathit{reaches}}(h, \sigma(v), \iota')))$ **:** This contradicts our assumption (*Eq.* 7).
  - $(\exists u', v.(\boldsymbol{\mathit{free}}(\Gamma(u')) \wedge \boldsymbol{\mathit{reaches}}(h, \sigma(u), \sigma(u')) \wedge \boldsymbol{\mathit{reaches}}(h, \sigma(v), \iota''))$ **:** Then we are done, picking u to be v.

We can similarly show (**Goal 21**) and (**Goal 20**) by case analyses, using (**Goal 22**). The argument for (**Goal 13**) is also similar, but is more involved (and is central to our treatment of constructors), and we show it in detail here:

To show (**Goal 13**), assume that for some variable v we have $\mathit{committed}(\Gamma(v))$. We need to show that $\mathit{deep\_init}(h', \sigma'(v)) \wedge \forall w.(\mathit{free}(\Gamma(w)) \Rightarrow \neg \mathit{reaches}(h', \sigma'(v), \sigma'(w)))$ holds. We consider two cases:

($v \neq x$) **:** Then $\sigma'(v) = \sigma(v)$ (by (*Eq.* 75)). To show $\mathit{deep\_init}(h', \sigma'(v))$ we assume (for a contradiction) that there is some address ι'' such that $\mathit{reaches}(h', \sigma'(v), \iota'') \wedge \neg \mathit{init}(h', \iota'')$ holds. Then, by (**Goal 18**), $\iota'' \in \mathit{dom}(h)$. We can then use (**Goal 22**) to give us three cases:

($\boldsymbol{reaches}(h, \sigma(v), \iota'')$) **:** By (**Goal 17**), we have $\neg \mathit{init}(h, \iota'')$. This contradicts our assumption (*Eq.* 7).

($\exists u.(\boldsymbol{committed}(\Gamma(u)) \wedge \boldsymbol{reaches}(h, \sigma(u), \iota''))$) **:** This also contradicts our assumption (*Eq.* 7).

($\exists u.(\boldsymbol{free}(\Gamma(u)) \wedge \boldsymbol{reaches}(h, \sigma(v), \sigma(u)))$ **:** This also contradicts our assumption (*Eq.* 7).

Thus, by contradiction, we can conclude $\mathit{deep\_init}(h', \sigma'(v))$. Now, suppose for a contradiction that, for some w we have $\mathit{reaches}(h', \sigma'(v), \sigma'(w)) \wedge \mathit{free}(\Gamma(w))$. We consider two cases:

($w \neq x$) **:** Then $\sigma'(w) = \sigma(w)$. We apply (**Goal 22**) to get three cases:

($\boldsymbol{reaches}(h, \sigma(v), \sigma(w))$) **:** This contradicts our assumption (*Eq.* 7).

($\exists u.(\boldsymbol{committed}(\Gamma(u)) \wedge \boldsymbol{reaches}(h, \sigma(u), \sigma(w)))$) **:** This also contradicts our assumption (*Eq.* 7).

($\exists u.(\boldsymbol{free}(\Gamma(u)) \wedge \boldsymbol{reaches}(h, \sigma(v), \sigma(u)))$ **:** This also contradicts our assumption (*Eq.* 7).

($w = x$) **:** Then $\sigma'(w) = \iota_1$. We apply the version of (*Eq.* 22) from our I.H. (using (*Eq.* 80)) to get three cases (note that $\iota_1 \in \mathit{dom}(h_1)$):

($\boldsymbol{reaches}(h_1, \sigma(v), \iota_1)$) **:** This contradicts our assumptions about *alloc* (Definition 8).

($\exists u.(\boldsymbol{committed}(\Gamma'(u)) \wedge \boldsymbol{reaches}(h_1, \sigma_1(u), \iota_1))$) **:** This contradicts point 3 of $\Gamma'; \Delta' \vdash h_1, \sigma'$ (justified above), since $\mathit{free}(\Gamma'(\mathsf{this})) \wedge \sigma'(\mathsf{this}) = \iota_1$ holds.

($\exists u.(\boldsymbol{free}(\Gamma'(u)) \wedge \boldsymbol{reaches}(h, \sigma(v), \sigma_1(u)))$) **:** By our observations above, this implies that, for some u' we have $\mathit{free}(\Gamma(u')) \wedge \sigma(u') = \sigma_1(u)$. This contradicts our assumption (*Eq.* 7).

($v = x$) **:** Then by assumption, we have $\mathit{committed}(\Gamma(x))$. By Lemma 2(2) and (*Eq.* 86), we have $k = 1$. By (*Eq.* 85), we have $\mathit{committed}(T_i)$. Suppose for a contradiction that we have, for some address ι_2, $\mathit{reaches}(h', \iota_1, \iota_2) \wedge \neg \mathit{init}(h', \iota_2)$. Then, by (**Goal 18**), we have $\iota_2 \in \mathit{dom}(h_1)$. We can then apply the version of (*Eq.* 22) from our I.H. (using (*Eq.* 80)) to give three cases:

($\boldsymbol{reaches}(h_1, \iota_1, \iota_2)$) **:** By our assumptions about *alloc* (Definition 8) we must have $\iota_2 = \iota_1$. But, we have already proved (when showing (**Goal 18**)) that $\mathit{init}(h', \iota_1)$ holds — a contradiction.

($\exists u.(\boldsymbol{committed}(\Gamma'(u)) \wedge \boldsymbol{reaches}(h_1, \sigma'(u), \iota_2))$) **:** This contradicts point 3 of $\Gamma'; \Delta' \vdash h_1, \sigma'$ (justified above).

($\exists u, u'.(\boldsymbol{free}(\Gamma'(u)) \wedge \boldsymbol{reaches}(h_1, \iota_1, \sigma_1(u)) \wedge \boldsymbol{reaches}(h_1, \sigma_1(u'), \iota_2))$ **:** By definition of σ_1, either $u' = \mathsf{this}$ (in which case we argue as in the first of these three cases) or $u' = x_i$ for some i. However, since $\mathit{committed}(T_i)$ holds, by Lemma 2(2) we must have $\mathit{committed}(\Gamma'(x_i))$. This therefore contradicts point 3 of $\Gamma'; \Delta' \vdash h_1, \sigma'$ (justified above).

This justifies (by contradiction) that $\mathit{deep\_init}(h', \iota_1)$ holds. Finally, assume for a contradiction that, for some w we have $\mathit{free}(\Gamma(w)) \wedge \mathit{reaches}(h', \iota_1, \sigma(w))$. By similar case analysis using the version of (*Eq.* 22) from our I.H., we obtain that $\sigma(w)$ must have been reachable from σ_1, which is impossible since all initialised variables are newly-allocated or of committed type.

(**Cast**) **:** This case is analogous to the (VARASS) case, but works as if the effective type of the variable assigned from were the combination of the type of the cast and the initialisation expectation of the variable's declared types. The extra runtime check provides the necessary extra information about the value assigned.

(**CastBad**) **:** This case contradicts the assumptions of our lemma, since we must have $\epsilon = \mathsf{castExc}$.

(**Seq**) **:** From the form of the rule, we have:

$$s = (s_1; s_2) \tag{93}$$

$$\mathsf{ok}, h, \sigma, s_1 \rightsquigarrow h_1, \sigma_1, \mathsf{ok} \tag{94}$$

$$\mathsf{ok}, h_1, \sigma_1, s_2 \rightsquigarrow h', \sigma', \epsilon \tag{95}$$

From the rule (TSEQ) we additionally obtain:

$$\Sigma = \Sigma_1 \cup \Sigma_2 \tag{96}$$

$$\Gamma; \Delta \vdash s_1 \mid \Delta_1; \Sigma_1 \tag{97}$$

$$\Gamma; \Delta_1 \vdash s_2 \mid \Delta_2; \Sigma_2 \tag{98}$$

From our assumptions, we can immediately apply induction using (*Eq.* 97), and from this I.H. we can then apply induction a second time using (*Eq.* 98). This gives us two induction hypotheses with which to derive our goals.
The following goals follow directly from the I.H. for s_2: (**Goal 9**), (**Goal 12**), (**Goal 13**), (**Goal 10**), (**Goal 11**), (**Goal 14**). The following follow by direct combination of the corresponding properties in the two I.H.s for s_1 and s_2: (**Goal 15**), (**Goal 17**), (**Goal 18**), (**Goal 19**), (**Goal 21**), (**Goal 22**). This leaves only the following to justify: (**Goal 16**) and (**Goal 22**).
To show (**Goal 16**), we need only combine the corresponding properties from the two I.H.s, and then apply (**Goal 17**).
To show (**Goal 22**) requires the only substantial work of this case. To show this, suppose that we have addresses $\iota_1, \iota_2 \in \mathit{dom}(h)$ such that $\mathit{reaches}(h', \iota_1, \iota_2)$ holds. We must the justify that one of the three cases enumerated in the conclusion of (**Goal 22**) hold. Using our I.H. for s_2, we can apply the corresponding version of (*Eq.* 22) to obtain three cases (relating the final heap h' with the intermediate heap h_1):

($\boldsymbol{\mathit{reaches}}(h_1, \iota_1, \iota_2)$) **:** Then we can apply the corresponding version of (*Eq.* 22) to obtain three cases, each of which match the cases of (**Goal 22**).

($\exists u.(\boldsymbol{\mathit{committed}}(\Gamma(u)) \wedge \boldsymbol{\mathit{reaches}}(h_1, \sigma_1(u), \iota_2))$) **:** Then, by the version of (**Goal 20**) from the I.H. for s_1, there exists a variable v such that $\mathit{committed}(\Gamma(v)) \wedge \mathit{reaches}(h, \sigma(v), \iota_2)$, matching the second case for (**Goal 22**).

($\exists u, v.(\textbf{\textit{free}}(\Gamma(u)) \wedge \textbf{\textit{reaches}}(h_1, \iota_1, \sigma_1(u)) \wedge \textbf{\textit{reaches}}(h_1, \sigma_1(v), \iota_2))$ **:** Then, by the version of (**Goal 21**) from the I.H. for s_1, there exists a variable v with $\mathit{free}(\Gamma(v)) \wedge \mathit{reaches}(h, \iota_1, \sigma(v))$. By (**Goal 19**) there exists a v'' such that $\mathit{reaches}(h, \sigma(v''), \iota_2)$ holds.

(**SeqBad**) **:** By I.H., provided that $\epsilon \neq \mathsf{castExc}$ (from our assumptions), we know that $\epsilon = \mathsf{ok}$. Therefore, we have a contradiction with the premises of this rule.

□

4 Comparisons

In this section we compare our type system to the two most expressive systems in the literature, delayed types [5] and masked types [14]. For this purpose, we re-cast the main examples from these papers in our system and discuss the differences. Note that neither example could be handled by the techniques termed "Inflexible" in our introduction.

4.1 Delayed Types

Fig. 8 shows the running example from the delayed types paper. We took the liberty of slightly adapting the syntax to be closer to the notation used in the rest of our paper (and in languages such as Java and C#).

All constructors are generic over a time t, which lies in the future ($\mathsf{t} > \mathsf{Now}$). The receivers of the constructors are delayed with type t (by the declaration $\diamond\mathsf{t}$). Note that in the List constructor, the time t is used to instantiate the Node constructor ctor1. This expresses that the list and its sentinel node become fully initialised at the same time, which allows them to safely refer to each other before being fully initialised. Note that all constructors declare an initialisation effect, which could be avoided by forcing all constructors to initialise all non-null fields of their receiver. The delay-block in method insertAfter introduces a delay time t and asserts that by the end of the block, all objects with delay t are fully initialised.

Fig. 9 shows the same example in our system. In our system, the receivers of constructors are implicitly free; we do not have to introduce a delay parameter, and constructors initialise all non-null fields, meaning that we do not have to declare initialisation effects. Moreover, our system determines automatically when a new object may be referred to via committed references, which avoids delay-blocks. Consequently, we need only a single annotation in the whole example: on the parameter of the first Node constructor to indicate that the argument object need not be committed. We can achieve this by declaring the parameter parent as either free or unclassified. We chose the latter because it makes the constructor signature more general.

The comparison of the two versions of the example illustrates that our system not only reduces the amount of necessary annotations, but especially their complexity. Our system requires neither genericity over delays nor constraints on the time when an object becomes fully initialised.

```
class List {
  Node! sentinel ;
  ctor[t,t>Now] ◇t ()
    {this.sentinel}
  {
    Node!◇t tmp = alloc Node[t];
    tmp.ctor1[t](this);
    this.sentinel = tmp;
  }

  void insert (Object? data) {
    this.sentinel.insertAfter(data);
  }
}

class Node {
  List! parent; Node! prev; Node! next;
    Object? data;
    // for sentinel construction
    ctor1[t,t>Now] ◇t (List!◇t parent)
      {this.parent , this.prev, this.next}
    {
      this.parent = parent;
      this.prev = this;
      this.next = this;
    }
    // for data node construction
    ctor2[t,t>Now] ◇t (Node! prev, Node! next, Object? data)
      {this.parent, this.prev, this.next, this.data}
    {
      this.parent = prev.parent ;
      this.prev = prev;
      this.next = next;
      this.data = data;
    }
    void insertAfter (Object? data)
    {
      delay t {
        Node! newNode = alloc Node[t];
        newNode.ctor2(this, this.next, data);
      }
      this.next.prev = newNode;
      this.next = newNode;

    }
}
```

Fig. 8. Doubly-linked list with delayed types.

```
class List {
  Node! sentinel ;
  List () { this.sentinel = new Node(this); }

  void insert (Object? data) {
    this.sentinel.insertAfter(data);
  }
}

class Node {
  List! parent; Node! prev; Node! next;
    Object? data;
    // for sentinel construction
    Node([Unclassified]List! parent) {
      this.parent = parent;
      this.prev = this;
      this.next = this;
    }
    // for data node construction
    Node(Node! prev, Node! next, Object? data) {
      this.parent = prev.parent ;
      this.prev = prev;
      this.next = next;
      this.data = data;
    }
    void insertAfter (Object? data) {
      Node newNode = new Node(this, this.next, data);
      this.next.prev = newNode;
      this.next = newNode;
    }
}
```

Fig. 9. Doubly-linked list with free/committed types.

4.2 Masked Types

Fig. 10 shows the running example from the masked types paper. Our only adaptation to the class structure is to make class Node abstract, which reflects its use in the example and does not affect the typing. Masked types require extensive annotations to declare which fields of an object are not yet initialised. For instance, the constructor of class Leaf declares that the parent field may not be initialised when the constructor terminates. Similar declarations occur in the signature of the Binary constructor and the client code.

```
abstract class Node {
  Node! parent;
}

class Leaf extends Node {
   Leaf() effect *! -> parent! { }
}

class Binary extends Node {
  Node left, right;
  Binary(Node\parent!\Node.sub[l.parent] -> *[this.parent] l,
    Node\parent!\Node.sub[r.parent] -> *[this.parent] r)
    effect *! -> parent!, left[this.parent], right[this.parent] {
    this.left = l;
    this.right = r;
    l.parent = this;
    r.parent = this;
  }
}

Leaf\parent! l = new Leaf();
Leaf\parent! r = new Leaf();
Binary\parent!\left[root.parent]\right[root.parent]
  root = new Binary(l, r);
root.parent = root; // Now root has type (fully initialised) Binary.
```

Fig. 10. Tree with back-pointers using masked types.

Our version of the tree example is shown in Fig. 11. To be able to type the example, we had to make two minor changes. First, the original example initialises the parent of the new Binary object outside the constructor (last line of Fig. 10). Since our system enforces that constructors initialise all non-null fields, we moved the assignment to parent into both constructors. Second, the Binary constructor in the original example assigns this to the parent field of the arguments l and r. Our system does not permit these assignments because this is

free inside the constructor and, thus, may be assigned only to fields of other free objects. However, l and r are not free in this context. To avoid forcing the client to assign to these fields, one could provide a wrapper method createBinary(l, r), which calls the Binary constructor and afterwards sets the parent field of l and r. One might consider extending our approach to allow a point in the constructor body to be syntactically marked, indicating that initialisation is finished and the receiver can be "converted" to a committed reference early. However, this does not extend well to subclasses (a feature which our approach does not handle in detail, but which can be handled with an extension) - in this case it would not be sound to assume that initialisation was really complete, since subclass constructors might not have been run yet. An alternative solution would be to adopt Spec#'s non-delayed constructors [11], which force non-null fields to be initialised *before* calling the base constructor. After the call to the base constructor, the receiver may be assumed to be fully initialised and, thus, assigned to fields of committed objects.

The comparison of the two versions of the example illustrates that our system reduces the annotation overhead tremendously. By distinguishing only between free and committed references, we avoid the overhead of declaring which fields may be assumed to be initialised, and by requiring constructors to initialise all non-null fields, we avoid the effect-clauses for constructors. In fact, we do not have to add a single annotation to handle the initialisation of the cyclic structure. However, our system does not support the deferral of initialisation until after the constructor has terminated. If such a deferred initialisation is required, the constructor needs to assign a dummy object to the non-null field, which gets replaced later (similar dummy assignments are sometimes necessary for local variables in Java and C# to pass the definite assignment checks).

In our introduction, we claimed that the annotation overhead in the Masked Types system is much higher than with our own. In fact, there are two important points here. Firstly, and perhaps most importantly, our type system is conceptually extremely simple to learn. There are only two annotations (plus the default, committed), which have semantics which are intuitively simple to understand, and the rules for type inference for expressions/assignments are very simple. Conversely, the masked types syntax for annotations includes, amongst other features, grammars for flexible effects annotations and sequenced masks, incorporating syntaxes for paths in the heap, patterns, etc. The concepts and notations a programmer must learn to understand and use this type system are both more numerous and more advanced in nature. Furthermore, fully understanding the typing rules is very subtle - e.g., for eliminating field masks: "In general, if some dependencies form a strongly connected component in which no mask depends on a mask outside the component, they can all be removed together".

Secondly, while it is the case that in many simple cases, the *defaults* chosen by the Masked Types system can (like those of our system) mean that most if not all annotations can be avoided, it is in the more interesting cases (mutual/cyclic initialisation) that the technical differences between the proposals are apparent.

```
abstract class Node {
  Node! parent;
}

class Leaf extends Node {
   Leaf() { parent = this; }
}

class Binary extends Node {
  Node! left, right;
  Binary(Node! l, Node! r)
  {
    this.left = l;
    this.right = r;
    this.parent = this;
  }

}

Leaf! l = new Leaf();
Leaf! r = new Leaf();
Binary! root = new Binary(l, r);
l.parent = root;
r.parent = root;
```

Fig. 11. Tree with back-pointers and free/committed types.

The tree example above is the main such example in their paper ("Figure 3 shows how to safely initialize a binary tree with parent pointers"). The Masked Types defaults do not apply to this example - all of the annotations shown are necessary in that system. As we show explicitly above, our technique compares very favourably in terms of annotation overhead for such a data structure. The program code does need to be refactored slightly for use in our system because of certain design decisions in our work (comparable implementation tweaks are discussed for other examples in section 6.1 of the Masked Types paper), but the result is far shorter and simpler, requiring no annotations at all to deal with initialisation.

We believe that these differences illustrate different motivations. The Masked Types work provides a very general and expressive solution (which can definitely handle more-complex typing disciplines with regard to initialisation than our proposal), which also tackles alternative problems such as object recycling, which our paper does not. Our motivation, on the other hand, is very much to keep to a proposal which is as simple and lightweight as possible for programmers to be able to use for the specific problem of object initialisation.

5 Further Related Work

While we have already elaborated on the most-closely related work in our introduction, we discuss here various other relevant research.

Haack and Poll present a type system for object immutability [7], which has some similarities to our work. As they remark in Section 4 of their paper, the initialisation problem for immutability is considerably simpler because one does not need to handle complex interactions between immutable and mutable references, unlike the problems of initialising mutual or cyclic data structures with non-null types. The same is true for the Javari work of Tschantz and Ernst [15].

Various implementations and practical works have been based on the original proposals of Fähndrich and Leino [6]. These include Ekman and Hedin [4] (a pluggable types implementation), Hubert et al. [9] (a machine-checked analysis for inferring non-null types) and Hubert [8] (extending this result to the bytecode level). Male et al. [12] also present a bytecode verification for non-null types, while Chalin and James [1] present an empirical study on the use (and defaults) of non-null types. All of these works take essentially the original "raw types" approach of Fähndrich and Leino (if any) to object initialisation; that is, they cannot handle examples involving mutual or cyclic initialisation (with the exception of some special cases for the "this" reference in the work of Hubert et al.)

6 Conclusions

We have presented and formalised our type-based approach to sound and flexible object initialisation and non-nullity. While the formulation and proof of

our soundness result are complicated, the resulting type system is very simple. There are three aspects which make our soundness proof challenging. One is the extensive use of reachability in our type invariants. The second is our desire to maintain a standard operational semantics (no runtime support for object initialisation). The third is the formulation of our proof at the level of single local stack frames - we never look at the whole stack to express invariants such as "committed references cannot be reached from free references", which means hard work when switching stack frames. These three aspects all however make clear that the understanding required to *use* our system does not require the programmer to think in terms of multiple stack frames, to have a non-standard runtime in mind, or to be concerned with how deeply our type invariants can be depended on in the heap.

In the proof, we had to consider properties of arbitrary objects and their reachability relations. However, the guarantees and checks that a programmer *using* the type system deals with, are all phrased in terms of the types of variables in scope at any particular program point. The programmer never needs to consider properties like "some other reference in the heap reaches this object", but may rely on the type system to preserve the validity of configurations even across method boundaries. This, along with the generally low annotation overhead of our system, makes our solution usable by programmers.

While we have not yet implemented our system, there are indications that the system is expressive and requires little annotation overhead:

- Firstly, we have encoded the running examples of the Delayed Types and Masked Types papers in our system (see Section 4). The doubly-linked list (Fig. 9) requires a single annotation, the binary tree with back-pointers (Fig. 11) does not require any annotations.
- Secondly, the annotation overhead in our system is identical to the overhead in Spec#'s version of Delayed Types (each [Delayed] annotation in Spec# corresponds to either a free or unclassified annotation in our system). The largest application written with Spec#'s Delayed Types is the Spec#-verifier (SscBoogie); it requires only six [Delayed] annotations in over 20,000 lines of code; three constructors would have to be slightly adapted. Our system requires an equally low overhead, but is sound.
  This low overhead is achieved by making variables committed by default and only requiring explicit annotations for free and unclassified variables. While we make all non-null annotations explicit in our examples, we would reduce the overhead further by using non-null (!) as the default.

We plan to implement our type system in a version of Spec#, so that we can directly experiment with further code. Since Spec# is a superset of C#, we will also then be able to experiment with annotating existing C# application code.

Our formalisation is based on a very small language, and we sketch here how it can be extended to more language features:

Control and Data Flow. Adding conditionals and loops to the language does not change the basic type system. At join points, we must intersect the sets that track definite assignment. Control flow is much more interesting when

mixed with a dataflow analysis; after an if-test for x! = null we can *change* the type of local variable x to a non-null type, for the duration of the block (or until x gets assigned to). This use of dataflow analysis is common in existing implementations of non-null type systems, and makes the system much more usable to the programmer by avoiding too many "obvious" casts.

Generics. We can support Java-like generics by adopting the solution from Spec#; generic types may quantify over the class and non-nullity of a type, but it is simpler for its initialisation expectation not to be varied with the generic instantiation. This does not cause a loss of expressiveness; we can also support genericity over initialisation expectations, as described in Section 2.5.

Super Calls. The most relevant language feature which our paper omits is super() calls in constructors. This feature allows object initialisation to be spread across the constructors of all superclasses of a new object. We can extend our approach to handle this feature in the same way as the raw types approach does—by introducing variants of our "free" references which indicate that (local) initialisation has been completed at least as far as a specified superclass. Note that we do not need to prevent dynamic method calls on free references (a common source of initialisation errors); our type system will check that such calls expect free receivers, which will in turn force the method implementation not to assume that the receiver is initialised.

Concurrency. While we have proved the soundness of our system in a sequential setting, we are convinced that soundness carries over to concurrency. The key idea is to enforce that each object is thread-local until it is fully initialised. That is, we maintain an invariant that any shared object (reachable from more than one thread) is committed and, thus, fully initialised. For example, in Java this invariant can be maintained by two rules. Firstly, only committed Thread objects can be started, that is, the Thread.start method requires a committed receiver. This rule ensures that starting a new thread preserves the invariant because the Thread object cannot store a free reference. Secondly, only committed references can be stored in static fields. This rule ensures that threads cannot pass free references from one thread to another via a static field. Since starting a thread is a "synchronization action" in Java's memory model, this argument also applies to Java's weak memory model.

Acknowledgements. We would like to thank Manuel Fähndrich for helpful discussions of delayed types and the Spec# implementation. We thank Hermann Lehner for some useful discussions on the details of the formalisation. We particularly thank Arsenii Rudich for many discussions on the inception and details of this work, and especially for last-minute food supplies.

References

1. Chalin, P., James, P.R.: Non-null references by default in java: Alleviating the nullity annotation burden. In: ECOOP. pp. 227–247 (2007)
2. Chalin, P., Rioux, F.: Non-null references by default in the Java Modeling Language. In: SAVCBS. p. 9. ACM (2005)

3. ECMA-367: Eiffel analysis, design and programming language. ECMA (2006), 2006
4. Ekman, T., Hedin, G.: Pluggable checking and inferencing of non-null types for Java. Journal of Object Technology 6(7) (2007)
5. Fähndrich, M., Xia, S.: Establishing object invariants with delayed types. In: OOPSLA. pp. 337–350. ACM (2007)
6. Fähndrich, M., Leino, K.R.M.: Declaring and checking non-null types in an object-oriented language. In: OOPSLA. pp. 302–312. ACM (2003)
7. Haack, C., Poll, E.: Type-based object immutability with flexible initialization. In: Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming. pp. 520–545. Genoa, Springer-Verlag, Berlin, Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-03013-0_24
8. Hubert, L.: A non-null annotation inferencer for java bytecode. In: Proceedings of the 8th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering. pp. 36–42. PASTE '08, ACM, New York, NY, USA (2008), http://doi.acm.org/10.1145/1512475.1512484
9. Hubert, L., Jensen, T.P., Pichardie, D.: Semantic foundations and inference of non-null annotations. In: FMOODS. pp. 132–149 (2008)
10. Kogtenkov, E., Stapf, E.: Avoid a void: The eradication of null dereferencing (2009), draft paper in honor of Tony Hoare's 75^{th} birthday.
11. Leino, K.R.M., Müller, P.: Using the Spec# language, methodology, and tools to write bug-free programs. In: Advanced Lectures on Software Engineering—LASER Summer School 2007/2008. LNCS, vol. 6029, pp. 91–139. Springer (2010)
12. Male, C., Pearce, D.J., Potanin, A., Dymnikov, C.: Java bytecode verification for @nonnull types. In: Proceedings of the Joint European Conferences on Theory and Practice of Software 17th international conference on Compiler construction. pp. 229–244. CC'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg (2008), http://portal.acm.org/citation.cfm?id=1788374.1788395
13. Meyer, B.: Attached types and their application to three open problems of object-oriented programming. In: ECOOP. pp. 1–32 (2005)
14. Qi, X., Myers, A.C.: Masked types for sound object initialization. In: POPL. pp. 53–65 (2009)
15. Tschantz, M.S., Ernst, M.D.: Javari: adding reference immutability to java. SIGPLAN Not. 40, 211–230 (October 2005), http://doi.acm.org/10.1145/1103845.1094828