Freedom Before Commitment

A Lightweight Type System for Object Initialisation

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

Abstract

One of the main purposes of object initialisation is to establish invariants such as a field being non-null or an immutable data structure containing specific values. These invariants are then implicitly assumed by the rest of the implementation, for instance, to ensure that a field may be safely dereferenced or that immutable data may be accessed concurrently. Consequently, letting an object escape from its constructor is dangerous; the escaping object might not yet satisfy its invariants, leading to errors in code that relies on them. Nevertheless, preventing objects entirely from escaping from their constructors is too restrictive; it is often useful to call auxiliary methods on the object under initialisation or to pass it to another constructor to set up mutually-recursive structures.

We present a type system that tracks which objects are fully initialised and which are still under initialisation. The system can be used to prevent objects from escaping, but also to allow safe escaping by making explicit which objects might not yet satisfy their invariants. We designed, formalised and implemented our system as an extension to a non-null type system, but it is not limited to this application. Our system is conceptually simple and requires little annotation overhead; it is sound and sufficiently expressive for many common programming idioms. Therefore, we believe it to be the first such system suitable for mainstream use.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Classes and Objects

General Terms Design, Languages, Reliability

1. Introduction

Object-oriented programs maintain numerous invariants about their heap data structures. These invariants reflect design decisions that are crucial for developing, understanding, and maintaining the code. They are assumed by method implementations, for instance, to ensure that a field may be safely dereferenced.

Most invariants do not hold for newly-allocated objects; they need to be established during object initialisation before the code operating on the object may rely on them. Mainstream programming languages such as Java, C# and C++ provide constructors to separate initialisation code that has to establish invariants from other code that may rely on the invariants. A problem occurs when an object escapes from its constructor before it is fully initialised. The escaping object might not yet satisfy its invariants, which may lead to errors in code that relies on them. Due to dynamic method binding, determining which code potentially operates on an escaped object is in general non-modular.

Escaping occurs if a constructor (1) calls a method on the object under initialisation, (2) passes the object as an argument to a method or constructor, or (3) stores the object in a field of another object, in a static field, or in an array.

Letting an object escape from its constructor is often considered bad programming practice. Many programming guidelines and blogs recommend to avoid escaping, style checkers such as PMD [1] issue warnings for some forms of escaping, and languages such as Java and C# enforce some ad-hoc rules to prevent some forms of escaping (for instance, Java does not allow one to refer to the `this` literal before calling the superclass constructor). However, none of these approaches effectively prevents escaping.

Entirely preventing objects from escaping their constructors would be too restrictive. It is useful to call auxiliary methods on a new object, and to pass it to other constructors to set up mutually-recursive structures. All three forms of escaping occur in the Java API implementation. For instance, a constructor of class `LinkedList` calls the method `addAll` to add all elements of a collection to the new list; a constructor of class `ScrollPane` passes `this` as argument to a constructor of `PeerFixer`, which stores it in a field.

In this paper, we present a type system that tracks which objects are fully initialised and which are still under initialisation. The type system can be used to prevent objects from

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

OOPSLA'11, October 22–27, 2011, Portland, Oregon, USA.
Copyright © 2011 ACM 978-1-4503-0940-0/11/10. . . $10.00

escaping, but also to allow safe escaping by making explicit which objects might not yet satisfy their invariants. Our objective is to design a system that is suitable for mainstream use, which requires it to satisfy the following design goals:

1. *Modularity:* The type system can check each class separately from its subclasses and clients.
2. *Soundness:* The system is type safe: objects that are considered to be fully initialised do satisfy their invariants.
3. *Expressiveness:* The type system handles common initialisation patterns. In particular, it allows objects to escape from their constructors and supports the initialisation of cyclic structures.
4. *Simplicity:* The type system is conceptually simple and requires little annotation overhead, making it easy and convenient to learn and use.

For concreteness, we present our type system as an extension of a non-null type system [10], which has several advantages: (1) Preventing null-dereferencing statically is of great practical importance. (2) The expected invariant is very clear: fields of non-null types contain non-null values. (3) Targeting an invariant that can be checked by a type system rather than runtime checks or verification allows us to formalize the whole system in one coherent framework. (4) Most of the related work on object initialisation has been applied to non-null types, which enables detailed comparisons. Beyond non-nullness, our type system generally supports monotonic invariants: invariants that get established by a constructor and are never violated afterwards. These invariants include one-state invariants, which are supposed to hold in each execution state (such as non-nullness of a field), as well as two-state invariants, which are supposed to hold for all pairs of states (such as immutability of a data structure).

Contributions. The key contribution of this paper is a type system for object initialisation that is suitable for mainstream use. More specifically, we present:

- the first type system for object initialisation that meets all four design goals stated above;
- a formalization of the type system for a sequential core language with non-null types, and preservation proof;
- a discussion of how to support concurrency and additional language features such as arrays and static fields;
- a discussion of how to support monotonic invariants besides the non-nullity of fields;
- an implementation of the type checker in the Spec# compiler [15];
- an evaluation using two major applications written in Spec# as well as examples from the literature.

Outline. In the next section, we provide the background on non-null types, and discuss previous attempts to handle object initialisation in this context. We present the design of our type system informally in Sec. 3 and formalise it in Sec. 4. We discuss extensions of our type system to further language features and invariants in Sec. 5. We report on our implementation and its application in Sec. 6, discuss related work in Sec. 7, and conclude in Sec. 8.

2. Background on Non-Null Types

To detect null-dereferences statically, Fähndrich and Leino proposed a *non-null type system* [10], in which reference types can be annotated with non-nullity expectations. Their idea has been widely adopted in the research community—various non-null type systems have been developed for Spec# (an extension of C#) [11, 15], Eiffel [6, 17, 19], and Java [7]. In this section, we present those aspects of non-null types that we build on and summarise previous work on object initialisation in non-null type systems.

2.1 Non-Null Types

The existing non-null type systems share the same fundamental idea: 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. The type system enforces that expressions with non-null types do not evaluate to the null value; it then prevents null pointer exceptions by forbidding the dereferencing of expressions with possibly-null types.

The doubly-linked list example in Fig. 1 (the motivating example from [10]) illustrates the use of non-null types. Every `Node` has references to its predecessor and successor in the list. The corresponding fields `prev` and `next` are of the non-null type `Node!`, which means that the list is cyclic. In addition, each node has a non-null reference to the `List` object it belongs to. The list elements stored in the nodes are of type `Object?`, that is, are allowed to be null. Each instance of class `List` stores a non-null reference to a sentinel node. The method call `this.sentinel.insertAfter(data)` in method `insert` type-checks because `this` is implicitly non-null and `sentinel` is declared to be non-null; hence, both expressions may be dereferenced.

$C!$ 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 behaviour (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 never store null. Downcasts from possibly-null types to non-null types are possible and entail a runtime check.

2.2 Object Initialisation

The main technical challenge in designing a non-null type system is how to handle object initialisation. The problem

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

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

class Node {
  Node! prev; Node! next;
  List! parent;
  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 n = new Node(this, this.next, data);
    this.next.prev = n;
    this.next = n;
  }
}
```

Figure 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 Sec. 3.

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. Until all non-null fields of the newly-created object have been initialised with non-null values, it would not be sound to make use of their declared non-null information.

Several solutions have been proposed for tackling the problem of initialisation for a non-null type system. They all require constructors to initialise the non-null fields of their class with non-null values before they terminate[1]. This is enforced statically using a straightforward *definite assignment analysis*, which checks that each non-null field of a class can be statically guaranteed to be assigned to at least once in its constructor. The existing solutions differ in how they handle objects that escape from their constructor. We summarise the approaches in the following and evaluate them using the four design goals stated in the introduction.

Raw Types. The original work of Fähndrich and Leino [10] 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. An object is allowed to escape via a method or constructor call, provided the signature of the method or constructor explicitly permits its receiver or arguments to be raw (and consequently does not rely on their non-null guarantees). However, 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 prevents common implementations such as the mutual initialisation of multiple objects, or cyclic data structures such as the `Node` structure in the example from Fig. 1. So with respect to our design goals, Fähndrich and Leino's type system is relatively simple[2], sound, and modular, but not sufficiently expressive to handle the initialisation of recursive structures. The only work-around for this problem is to declare the fields of the recursive structure with possibly-null types and to inject a downcast each time they get dereferenced, which clutters up the code and leads to unnecessary runtime checks.

Delayed Types. Fähndrich and Xia's *delayed types* [11] 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. So with respect to our design goals, delayed types are expressive, sound, and modular. However, the complexity of the annotations makes the system as presented in the paper unsuitable for mainstream use.

Indeed, when implementing the system in Spec# [15], Fähndrich and Xia decided to greatly cut down the complexity of the type system, including only a single "Delayed"

[1] Masked Types [21] free constructors from this obligation for those fields that the constructor's signature declares to be left un-initialised.

[2] Some complexity comes from the fact that raw types include information to which class in the inheritance hierarchy an object has been initialised. For instance, `raw(A)` expresses that the fields declared in class `A` and its superclasses have already been initialised, whereas the remaining fields of the object might not.

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. This assumption can be exploited to provoke a null pointer exception; we discuss such an example in Sec. 7. 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.

Attached Types. Eiffel's non-null types (called "attached types") do not appear to address the problem of object initialisation soundly. According to the Eiffel standard [6], a field 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." Since objects may escape from their creation procedures (constructors), this is not sufficient for soundness. The problematic situation can sometimes be avoided by providing default creation procedures for all types of non-null fields—these get implicitly called when a field is found not to be initialised yet. However, default initialisation cannot handle cases such as cyclic lists, or the mutual initialisation of objects. So with respect to our design goals, attached types are simple, expressive, and modular, but not sound.

The actual Eiffel implementation appears (by experiment) to actually prevent unsoundness by enforcing much stronger rules: Using an object under initialisation as receiver or argument of a call is permitted if the code of the called method type-checks without making non-null assumptions for that object. For dynamically-bound methods, this check needs to be repeated for each override of the called method, which makes the type checking non-modular. Moreover, an object may not be assigned to any field until its initialisation is complete. This rule makes cyclic and mutual initialisations impossible. So the Eiffel implementation is simple and sound, but neither expressive nor modular.

Masked Types. The recent work of Qi and Myers [21] 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 is sound, modular, and the most expressive approach yet, it is unlikely that an average programmer would find it simple enough to handle the everyday problem of sound object initialisation. We provide a more detailed discussion and comparison with Masked Types in Sec. 7.2.

Summary. As summarised by the following table, none of the existing solutions for object initialisation in non-null type systems satisfies all four design goals that we consider essential for the usefulness of a type system. Delayed types and masked types are the only sound systems that are sufficiently expressive to handle recursive structures, but both systems are conceptually complex and require significant annotation overhead. Since our goal is to develop a system for mainstream use, we resolve the trade-off differently. As far as practical examples are concerned, our system is slightly less expressive than delayed types and masked types, but significantly simpler.

System	Simple	Expressive	Sound	Modular
Raw Types	✓	–	✓	✓
Delayed Types (paper)	–	✓	✓	✓
Delayed Types (implementation)	✓	✓	–	✓
Attached Types (ECMA)	✓	✓	–	✓
Attached Types (implementation)	✓	–	✓	–
Masked Types	–	✓	✓	✓

3. The Design

In this section, we explain the main concepts of our type system informally. We introduce initialisation types that reflect whether an object has been fully initialised or is still under initialisation, and motivate the most important type rules. The type system is formalized in the next section. Additional language features such as subclassing, arrays, generics, and concurrency are discussed later in Sec. 5.

3.1 Initialisation States

Each object is in one of two *initialisation states*: it is either *under initialisation* or it is *initialised*. When a new object is allocated as part of executing a new-expression, it is initially under initialisation until execution reaches a point from which on we consider the object to be initialised. This change of the initialisation state happens when *a certain* new-expression terminates, but not necessarily *the* new-expression that created the object. We call the point at which the state change occurs the *commitment point* of the object and will explain later when it occurs.

Initialised objects have to satisfy their invariants, in particular, their non-null fields must contain non-null values. Moreover, references stored in fields of an initialised object must point to objects that are also initialised. This *deep initialisation* guarantee is important for the practicality of the system. It ensures that all objects that are encountered while traversing an object structure starting from an initialised object are also initialised and, thus, the traversal can rely on the invariants of the whole structure.

Objects under initialisation might satisfy their invariants, but they are not required to. The fields of such objects may store references to objects in either initialisation state. These fields may refer to objects that are themselves under initialisation, which allows one to initialise cyclic structures.

It is important to understand that initialisation states are a purely conceptual notion. Neither do we store an object's initialisation state in memory nor is it generally possible to infer an object's initialisation state by inspecting the heap. In particular, an object might satisfy its invariants but nevertheless be under initialisation because it has not yet reached its commitment point. A program may still assign an uninitialised object to a field of such an object and thereby violate its deep initialisation. Such assignments are no longer possible once the object has passed its commitment point.

3.2 Initialisation Types

In our type system, the type of an expression reflects the initialisation state of the object the expression refers to at runtime. It uses this information to provide guarantees about the invariants of the object and to enforce restrictions that guarantee soundness. For this purpose, we equip each reference type with one of the following three *initialisation modifiers*:

committed: Expressions of committed types evaluate to references to initialised objects.

free: Expressions of free types evaluate to references to objects under initialisation.

unclassified: Expressions of unclassified types may evaluate to any reference. An unclassified type is a supertype of the corresponding committed and free types.

Note that these initialisation modifiers are independent of the non-nullity of a type—we can have both non-null and possibly-null types with any of the three modifiers above (in the latter case, guarantees about the "referred-to object" only apply if the reference is not null). Despite attaching both nullity and initialisation information to reference types, the annotation overhead of our system is low. Almost all references handled in a program are committed, non-null references, such that a suitable default avoids overhead for those references. Initialisation modifiers need to be declared explicitly only for non-trivial initialisation patterns. In our examples, we make nullity information explicit; the default initialisation modifier 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.

Fig. 2 illustrates the use of these modifiers. When a new object is allocated, its type is initially a free type (hence the default for `this` in constructors). Once it reaches its commitment point, the type changes to a committed type, indicating that the program can now rely on the object to be deeply initialised. The type system enforces that the object will remain deeply initialised until its de-allocation.

3.3 Fields

Field types include non-null annotations, but no initialisation modifiers. In particular, there are no "free fields" for

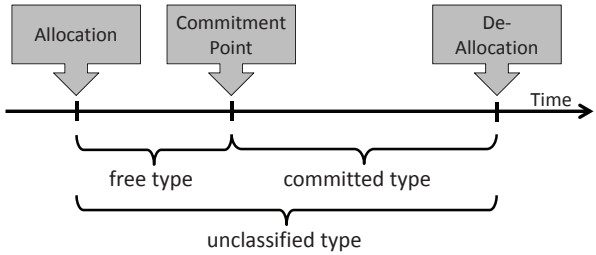

Figure 2. Newly-allocated objects have a free type until they reach their commitment point, when the type changes to a committed type for the rest of the object's lifetime. Unclassified types subsume free and committed types.

two reasons. First, the type of an object changes from free to committed when it reaches its commitment point. This type change would be problematic if the object was referenced from a free field because this field would be ill-typed after the type change. Second, free fields contradict the expectation that objects of committed types be deeply initialised.

Field Update. For a field update of the form $x.f = y$, our type system performs nullity and initialisation checks. The nullity checks are trivial: x must be non-null, and the nullity of y must conform to the nullity declared for f. The former check prevents null-pointer dereferencing whereas the latter ensures that only non-null values are assigned to non-null fields; in particular, this check ensures that the non-nullity invariant is monotonic. Once a non-null field has been assigned a non-null value, it will remain non-null.

For the initialisation types, the update is allowed if the initialisation modifiers satisfy at least one of the following two cases. First, if x is free, we may store objects with any initialisation state in its fields. This is acceptable because the free modifier does not make any guarantees about the initialisation state of reachable objects. In particular, allowing y to be free enables the initialisation of cyclic structures, as illustrated by `Node`'s first constructor (Fig. 1). All its field updates type check 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 state of reachable objects anyway.

The following table summarises the admissible field updates. The case where x is committed and y is free is disallowed because such an update would violate the deep initialisation requirement for committed objects. An update is allowed for unclassified references only if it is allowed for both committed and free references.

$x.f = y$		y		
		committed	free	unclassified
x	committed	✓	–	–
	free	✓	✓	✓
	unclassified	✓	–	–

An important consequence of the rule for field updates is that our type system must prevent aliasing between committed and free references, which we call *cross-type aliasing*. If an object x was reachable via a committed and a free reference then one could use the free reference to store an uninitialised (free) object in a field of x, which would clearly violate the deep initialisation expectation of the committed reference to x and therefore compromise soundness. Cross-type aliasing is prevented by not having a subtype relationship between committed and free types (in contrast to raw types, which are supertypes of the corresponding non-raw types [10]).

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 type is the only type 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 store both free and committed references. The following type summarises this rule.

$x.f$		f	
		$C!$	$C?$
x	committed	committed $C!$	committed $C?$
	free	unclassified $C?$	unclassified $C?$
	unclassified	unclassified $C?$	unclassified $C?$

Consider the new-expression in method `insertAfter` (Fig. 1). The type of the second argument `this.next` is committed and non-null because `this` is by default committed in methods, as discussed in the next section.

3.4 Methods and Constructors

Method signatures include initialisation modifiers for the method parameters as well as for the receiver. The type rule for method calls is like in all object-oriented languages: The types of the actual arguments must be subtypes of the declared parameter types. An analogous check is performed for the receiver of a call. This rule ensures that an object under initialisation may be passed to a method as receiver or argument only if the receiver or parameter in the method signature has a free or unclassified type. In both cases, the method will not rely on the object to be initialised, which makes this form of escaping safe. Method overriding requires the usual contra-variance of parameter types and co-variance of result types.

Constructors are treated analogous to methods, but their signature does not contain an initialisation modifier for the receiver because the receiver of a constructor is always free. The first constructor in class `Node` declares its parameter `parent` with a free-modifier. Therefore, the `List` constructor may pass the free object `this` as an argument.

A definite assignment analysis enforces that the body of a constructor establishes the invariant of the receiver, that is, assigns non-null values to all non-null fields declared in the enclosing class. In our example, both `Node` constructors satisfy this analysis because they initialise all three non-null fields.

Note that the definite assignment analysis is the reason why we require invariants to be monotonic, even for free objects. Assigning null to a non-null field of a free object would not compromise soundness because free objects are not expected to satisfy their invariants. However, if such non-monotonic updates were permitted, the definite assignment analysis would have to assume conservatively that each time a free object is passed to a method or constructor, it will come back with its fields set to null, even if they were previously assigned non-null values. Such an analysis would produce many false positives and, thus, not be practical.

3.5 Commitment Points

A central concept of our type system is the notion of commitment point, the point in time when an object is no longer under initialisation but now considered initialised; this change of the initialisation state will be reflected in a change of the object's type from free to committed. The commitment point may occur when two requirements are satisfied. First, the type system must be able to determine statically that the object is deeply initialised. Second, when an object reaches its commitment point and becomes committed, the type system must be able to guarantee that there are no free references to the object. As explained earlier, such cross-type aliases could be used to violate the deep initialisation of the object and, thus, compromise soundness.

To satisfy these requirements, we define the commitment point as follows. When a new-expression *that takes only committed arguments* terminates then all objects that have been created during the execution of this new-expression (and the associated constructor) reach their commitment point (unless they have already reached it when a nested new-expression with only committed arguments terminated).

Consider the `List` constructor in Fig. 1. The `Node` object n created by the expression `new Node(this)` does not yet reach its commitment point when the new-expression terminates because it takes a free argument, `this`. In fact, n is *not* deeply initialised then because n`.parent.sentinel` is still null and, therefore, violates the first requirement for commitment points. The same argument applies if an unclassified reference is passed as argument to a new-expression, since it may (via subtyping) disguise a free reference. But when the expression `new List()` terminates then both the new `List` object and its sentinel node reach their commitment point.

This definition of commitment point leads to a very simple type rule for new-expressions. An expression `new` C`(`$x_1, \ldots, x_n$`)` has type committed $C!$ if the static types of all actual arguments x_i are committed. Otherwise, it has type free $C!$. We justify our definition of commitment point and the corresponding type rule for new-expressions in the following.

New-Expressions with Free or Unclassified Arguments. The `List` constructor illustrates why it would be unsound to assign a committed type to new-expressions with at least one free or unclassified argument. However, considering the new object as free is safe if we can guarantee that there are no cross-type aliases. This is the case because no local variables refer to the new object and because inside the constructor, the new object was referred to via a free reference and, therefore, the constructor could not store the reference in the field of a committed object.

New-Expressions with Committed Arguments Only. Consider a new-expression where all arguments are committed (which subsumes the case that the constructor does not have parameters). Let n be the object that gets created by this new-expression. During the execution of n's constructor, the set of reachable objects consists of the set of objects R that are reachable from the new-expression's arguments and the set of objects N that includes n and all objects created during the execution of n's constructor. The situation is illustrated in Fig. 3. For the `List` constructor, the set R is empty, whereas N contains the new `List` object and its sentinel node.

When the new-expression terminates, the constructors of all objects in N have run and therefore, the non-null fields of these objects contain non-null values. The values assigned to their fields are references to objects in R or N because these are all the reachable objects. Therefore, we know that the objects in N are deeply initialised because all objects reachable from them are in R or in N and thus initialised[3]. So the first requirement for commitment points is satisfied.

To argue that the second requirement is satisfied, we have to show that there is no cross-type aliasing, that is, no free reference to an object in N. When the new-expression terminates, no local variable refers to an object in N. The only objects that possibly refer to objects in N are the objects in N and R. These objects are all committed when the new-expression terminates, and so is the reference to the new object n.

Note that for constructors that have only committed formal parameters, it might be tempting to consider the receiver initialised as soon as the definite assignment analysis confirms that all fields have been initialised. However, this solution would be unsound because it violates both requirements for commitment points. First, during the execution of a constructor, we can in general not determine modularly whether there are subclass constructors, which have not executed yet. So there might be non-null fields declared in subclasses that have not been initialised. Second, the constructor might have local variables that store free references to the receiver, which creates cross-type aliasing.

[3] This argument generalises trivially to global data if the global variables (that is, static fields) are enforced to store committed references, see Sec. 5.3.

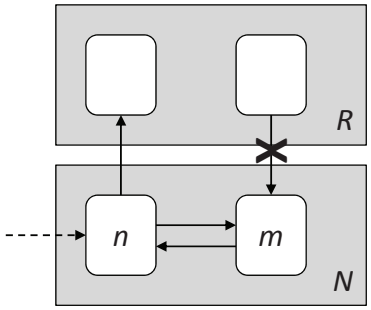

Figure 3. Heap structure for the execution of a new-expression that takes only committed arguments. Objects are depicted by rounded boxes; references in fields and on the stack are depicted by solid and dashed arrows, respectively. The set N contains the objects created during the execution of the new-expression; the set R contains the (committed) objects reachable from the new-expression's arguments. Here, both objects in N are free until the new-expression terminates and, therefore, cannot be referred to by the committed objects in R.

3.6 Dataflow Analysis

We use a simple intra-procedural dataflow analysis to refine the non-nullity and initialisation information provided by the type system. For instance, if a conditional-statement tests a local variable x for being non-null then x may be assumed to have a non-null type in the then-block of the conditional and, therefore, may be dereferenced and assigned to non-null variables. Similarly, a dataflow analysis can provide initialisation information for unclassified variables and fields of free objects. 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 largely ignored in the following. Any such analysis, whether simple or complex, could be used to complement the type system we outline. For example, if support is needed for initialising fields via method calls, suitably chosen extra annotations on method signatures and/or inter-procedural analyses could be used to provide the necessary extra information to the type checker.

3.7 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-casts to change initialisation modifiers (from unclassified to free or committed) nor do we allow instanceof-expressions to check initialisation modifiers. Such casts and instanceof-expressions would essentially require us to store each object's initialisation state in the heap, which leads to runtime overhead and problems with library code. Consequently, the only runtime support our system re-

quires is simple non-null checks for down-casts to non-null types.

4. 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 commitment points described in Sec. 3.5. Our soundness result (in Sec. 4.4) makes these arguments explicit.

4.1 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 flds() from classes to sets of field names, and a function meths() from classes to sets of method names.*
Non-null annotations*, ranged over by n, are defined by* $n ::= ? \mid !$
Initialisation modifiers*, ranged over by k, are defined by:*

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

Types*, ranged over by T, are defined by* $T ::= C^k n$.
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. Simple types are used in field declarations and in casts, where initialisation modifiers are not permitted.

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

DEFINITION 2 (Type Relations). Initialisation specialisation *is a binary relation on initialisation modifiers, written $k_1 \leq k_2$ and defined by: $k_1 \leq k_2 \Leftrightarrow k_1 = k_2 \vee k_2 = \diamond$.*
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:*
$C_1{}^{k_1} n_1 \leq C_2{}^{k_2} n_2 \Leftrightarrow C_1 \leq C_2 \wedge k_1 \leq k_2 \wedge n_1 \leq n_2$.

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 modifier for the *receiver* of the method, as well as its arguments and return type. Constructors only have such modifiers for arguments; during execution of a constructor its receiver is always a free reference, and after execution the initialisation type of the new expression is determined by those of the passed arguments (cf. Sec. 3.5). Both kinds of signatures also include declarations of local variables used within the method body, but this is just to simplify the formal presentation.

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.*
A Method Signature *is a four-tuple $(k, \overrightarrow{x_i{:}T_i}, T, \overrightarrow{y_j{:}T_j})$, whose elements are as follows[4]: (1) an initialisation modifier k, indicating the initialisation type of the receiver, (2) a sequence $\overrightarrow{x_i{:}T_i}$ of parameter names (variable names) along with their declared types, (3) a type T representing the return value of the method, and (4) a sequence $\overrightarrow{y_j{:}T_j}$ of local variable names along with their declared types.*
A Constructor Signature *is a two-tuple $(\overrightarrow{x_i{:}T_i}, \overrightarrow{y_j{:}T_j})$, 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 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.*

Our statements include assignments, method calls, object creation and casts. We do not include conditionals since they would only be of interest when combining our type system with a dataflow analysis (Sec. 3.6). 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 expressions can be decomposed using local variables.

[4] We use vector notation $\overrightarrow{x_i}$ for sequence/repetition with elements indexed by i (the index clarifies which terms vary in a sequence). Use of different indexes indicates sequences with different index sets.

DEFINITION 4 (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:*

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

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`*:*

s	$::=$	$x = e$	*(variable assignment)*
	$\mid$	$z.f = y$	*(field assignment)*
	$\mid$	$x = y.m(\overrightarrow{z_i})$	*(method call)*
	$\mid$	$x = \texttt{new}\ C(\overrightarrow{z_i})$	*(object creation)*
	$\mid$	$x = (t)y$	*(cast)*
	$\mid$	$s_1; s_2$	*(sequential composition)*

Note that casts employ only simple types. As discussed in Sec. 3.7, we do not support casts that change the initialisation type of a reference.

4.2 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 used in judgements, to (conservatively) track which variables can be safely read from. 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 5 (Static Type Assignment). *A* type environment Γ *is a partial function from program variables to 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, possibly reading variables in Δ. The judgements are defined in Fig. 4.*
Statement typing *is defined by judgements $\Gamma; \Delta \vdash s \mid \Delta'; \Sigma$, indicating that s is well-typed under assumptions Γ, reads only variables in Δ and, after execution, will guarantee that variables Δ' and fields Σ are definitely assigned. The judgements are defined in Fig. 5.*

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

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

Figure 4. Expression typing.

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

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

$$\frac{\Gamma; \Delta \vdash y : C^{k_1}! \quad \mathit{mSig}(C, m) = (k_2, \overrightarrow{x_i{:}T_i}, T, \overrightarrow{y_j{:}T_j}) \quad \Gamma; \Delta \vdash z_i : T_i' \quad T_i' \leq T_i \quad T \leq \Gamma(x) \quad k_1 \leq k_2}{\Gamma; \Delta \vdash x = y.m(\overrightarrow{z_i}) \mid \Delta \cup \{x\}; \emptyset}\ (\textsc{Tcall})$$

$$\frac{\mathit{cSig}(C) = (\overrightarrow{x_i{:}T_i}, \overrightarrow{y_j{:}T_j}) \quad \Gamma; \Delta \vdash z_i : T_i' \quad T_i' \leq T_i \quad k = \bigwedge \overrightarrow{\mathit{committed}(T_i')} \quad C^k! \leq \Gamma(x)}{\Gamma; \Delta \vdash x = \texttt{new}\ C(\overrightarrow{z_i}) \mid \Delta \cup \{x\}; \emptyset}\ (\textsc{Tcreate})$$

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

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

Figure 5. Statement typing.

Our expression typing judgement plays a dual role in our formalisation. It checks not only that the expression is typeable in a standard way, but also that it can be "read from"; any variable mentioned has to be known to be already

assigned. Since our judgement does not include subtyping (which is dealt with at the level of typing statements), we have the property that a variable x can be typed exactly with the type $\Gamma(x)$, and *only* when x occurs in Δ (i.e., it has been assigned). When defining subsequent typing rules, we choose between applying the rule (TVAR) to variables, or just looking up the type $\Gamma(x)$ directly, depending on whether or not we require the variable to already be assigned.

The rule (TVAR) looks up the type for a variable in the environment, also checking that the variable has already been assigned (i.e., it is named in Δ). The rule (TNULL) allows a `null` expression to be typed with any class and initialisation type, but of course it must take a possibly-null type. The rule for field read, (TFLD), first checks that the receiver expression x is typeable with a non-null type; this also implicitly enforces the requirement that x is known to be assigned (this would not have been the case if we had looked up the type for x directly in Γ). Then, based on the class type C of x, the appropriate field type is retrieved (this implicitly requires that the field is defined for the class). The type for the whole expression is then derived. The class type is whatever was declared in the field type. The type is committed if the receiver expression x was typed as committed, and is unclassified otherwise. Finally, the expression is non-null if the field was declared non-null and the receiver was also of committed type; otherwise, the expression is typed as possibly-null. Note that while we have no subtyping here, we build it into the statement typing rules where required.

In the rules for statement typing (Fig. 5), we take care to "update" the variables known to be assigned (via the second set of variables in the judgements), and also to record the fields of `this` which get assigned values (this information is only needed for checking constructor bodies, but for simplicitly, we accumulate it in general). The rule for variable assignment, (TVARASS), checks that the source expression is typeable with a subtype of the declared type of the variable, and adds the variable to those known to be assigned. Note that the destination variable is not subject to the expression typing judgement; this means we do not erroneously insist on it being assigned beforehand (while any variables referenced in the source expression must be). In the rule for field assignment, (TFLDASS), we require both the receiver and the source variable to be assigned and typeable, and impose the additional requirement that the receiver must be free or the source variable must be committed. In the case where the receiver is `this`, we also record the field to be assigned, in Σ.

The rule for method call (TCALL) checks that the receiver is assigned and non-null, and then checks that the types of the receiver, arguments and return value destination, all agree with the declared method signature. Note that in the case of the receiver, this involves checking that the receiver initialisation type k_1 specialises the initialisation type k_2 declared for the receiver in the method signature. Σ is empty in this rule; the basic flow analysis we use here does not handle initialisation via a method (but see Sec. 3.6).

The rule for object creation (TCREATE) specifies that the initialisation type of the returned value is defined to be committed (1) if all the arguments have committed types, and free (0) otherwise; we denote this "conjunction" with the shorthand $\bigwedge \overrightarrow{\mathit{committed}(T'_i)}$. Our rule for casting (TCAST) incorporates a variable assignment, and allows the class type and non-null type of the value to change arbitrarily, but does not affect the initialisation type. Finally, our rule for sequential composition (TSEQ) chains together the inference of which variables have been assigned, and accumulates the fields of `this` known to have been assigned in either statement.

If one wanted to adapt our formalisation to check for some other kind of invariant (such as immutability) and not for non-null types, the changes that would be needed are fairly small. Firstly, one should remove the requirement that variables dereferenced always have non-null types (TFLD, TFLDASS, TCALL). One might also simplify/modify the rule for computing the type of a field read (TFLD). The role of Σ in our judgements is specific to tracking progress towards a newly-constructed object being initialised. The use of this judgement for the proof might well vary depending on the invariant in question. Finally, the check for a well-formed constructor (WFCONS) below would need to be adjusted to guarantee initialisation with respect to the invariant in question (for non-null types, we require that every non-null-declared field of the new object gets assigned).

We can now type check class definitions.

DEFINITION 6 (Well-formed program). *A program is well-formed if for each class C of the program, each method $m \in \mathit{meths}(C)$ is well-formed ($\vdash_m C, m$), and the constructor is well-formed ($\vdash_C C$). These judgements are defined in Fig. 6.*

$$\frac{\begin{array}{c}\mathit{mBody}(C,m) = s \\ \mathit{mSig}(C,m) = (k, \overrightarrow{x_i{:}T_i}, T, \overrightarrow{y_j{:}T_j}) \\ \neg \mathit{nullable}(T) \Rightarrow \texttt{res} \in \Delta \\ \Gamma = (\overrightarrow{x_i{:}T_i}, \texttt{this}{:}C^k!, \overrightarrow{y_j{:}T_j}, \texttt{res}{:}T) \\ \Gamma; \{\overrightarrow{x_i}, \texttt{this}\} \vdash s \mid \Delta; \Sigma\end{array}}{\vdash_m C, m} \ (\textsc{wfMeth})$$

$$\frac{\begin{array}{c}\mathit{cBody}(C) = s \quad \mathit{cSig}(C) = (\overrightarrow{x_i{:}T_i}, \overrightarrow{y_j{:}T_j}) \\ \{f \in \mathit{flds}(C) \mid \neg \mathit{nullable}(\mathit{fType}(C,f))\} \subseteq \Sigma \\ \Gamma = (\overrightarrow{x_i{:}T_i}, \texttt{this}{:}C^0!, \overrightarrow{y_j{:}T_j}) \\ \Gamma; \{\overrightarrow{x_i}, \texttt{this}\} \vdash s \mid \Delta; \Sigma\end{array}}{\vdash_C C} \ (\textsc{wfCons})$$

Figure 6. Well-formed methods and constructors.

Essentially, every method body (WFMETH) must be typeable with respect to its signature, under the assumption

that all parameters are initially assigned. Furthermore, the method body must assign to the result variable `res` (of course, this restriction could be relaxed to support void methods). For constructors (WFCONS), the body must be typeable with respect to its signature, along with the assumption that the receiver is a free, non-null reference of the appropriate class type. Furthermore, every non-null-declared field of the class must be assigned a value in the method body.

4.3 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; in particular, no support for the initialisation aspects of our type system is needed at runtime; as we described earlier, initialisation states of objects are purely conceptual, and used to explain the workings of the static type system. This fact is essential for the feasibility of our type system for mainstream languages.

DEFINITION 7 (Heaps, Values and Allocation). *We assume a finite set of* addresses, *ranged over by* ι.
Values, *ranged over by* v *are defined*[5] *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/extension *(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:

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

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 8 (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 values* v *plus the special value* `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:*

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

We can now define our operational semantics.

DEFINITION 9 (Operational Semantics). Exception States, *ranged over by* ϵ*, are one of three possible concrete values:* $\epsilon ::= \texttt{ok} \mid \texttt{derefExc} \mid \texttt{castExc}$.
Runtime Type Assignment *assigns simple types to runtime values, according to the subclassing relationship in the program. It is defined in Fig. 7. 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. 8.*

$$\frac{}{h \vdash null : C?}\ (\text{RNULL}) \qquad \frac{cls(h, \iota) \leq C}{h \vdash \iota : C\ n}\ (\text{RADDR})$$

Figure 7. Runtime type assignment.

4.4 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 10 (Initialisation and Reachability). *An address is* locally initialised *in a heap, written* $init(h, \iota)$*, if all non-null fields contain non-null values:*

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

An address reaches *another address in a heap, written* $reaches(h, \iota_1, \iota_2)$*, as defined recursively by the least fixpoint solution of the following equation:*

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

Given an address and heap, the set of addresses reachable, *written* $reachable(h, \iota)$ *is defined by:* $reachable(h, \iota) = \{\iota' \mid reaches(h, \iota, \iota')\}$.
An address is deeply initialised *in a heap, written as a predicate* $deep\_init(h, \iota)$*, if all reachable addresses are locally initialised:*

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

[5] 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.

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

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

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

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

$$\frac{\begin{array}{c}\sigma(y) = \iota \quad C = \mathit{cls}(h, \iota) \\ \mathit{mSig}(C, m) = (k, \overrightarrow{x_i{:}T_i}, T, \overrightarrow{y_j{:}T_j}) \\ \sigma_1 = \texttt{this} \mapsto \iota, \overrightarrow{x_i \mapsto \sigma(z_i)}, \texttt{res} \mapsto \mathit{null}, \overrightarrow{y_j \mapsto \mathit{null}} \\ \mathit{mBody}(C, m) = s \quad \texttt{ok}, h, \sigma_1, s \ \rightsquigarrow \ h', \sigma', \epsilon \end{array}}{\texttt{ok}, h, \sigma, x = y.m(\overrightarrow{z_i}) \ \rightsquigarrow \ h', \sigma[x \mapsto \sigma'(\texttt{res})], \epsilon} \ (\textsc{call})$$

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

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

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

$$\frac{h \not\vdash \sigma(y) : t}{\texttt{ok}, h, \sigma, x = (t)y \ \rightsquigarrow \ h, \sigma, \texttt{castExc}} \ (\textsc{castBad})$$

$$\frac{\begin{array}{c}\texttt{ok}, h, \sigma, s_1 \ \rightsquigarrow \ h_1, \sigma_1, \texttt{ok} \\ \texttt{ok}, h_1, \sigma_1, s_2 \ \rightsquigarrow \ h_2, \sigma_2, \epsilon \end{array}}{\texttt{ok}, h, \sigma, s_1; s_2 \ \rightsquigarrow \ h_2, \sigma_2, \epsilon} \ (\textsc{seq})$$

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

Figure 8. Operational semantics.

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.

DEFINITION 11 (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. $\mathit{dom}(\sigma) = \mathit{dom}(\Gamma) \ \wedge \ \texttt{this} \in \mathit{dom}(\sigma)$
2. $\forall \iota \in \mathit{dom}(h), f \in \mathit{flds}(\mathit{cls}(h, \iota)) : \ (\iota, f) \in \mathit{dom}(h) \wedge (h(\iota, f) \neq \mathit{null} \Rightarrow \mathit{cls}(h, h(\iota, f)) \leq \mathit{fType}(\mathit{cls}(h, \iota), f))$
3. $\forall x \in \mathit{dom}(\sigma) : (\neg \mathit{nullable}(\Gamma(x)) \wedge x \in \Delta \Rightarrow \sigma(x) \neq \mathit{null})$
4. $\forall x \in \mathit{dom}(\sigma) : \ (\sigma(x) \neq \mathit{null} \Rightarrow h \vdash \sigma(x) : \Gamma(x))$
5. $\forall x, y \in \mathit{dom}(\sigma) : \ (\mathit{committed}(\Gamma(x)) \Rightarrow \mathit{deep\_init}(h, \sigma(x)) \wedge (\mathit{free}(\Gamma(y)) \Rightarrow \neg \mathit{reaches}(h, \sigma(x), \sigma(y))))$

We can now 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* $\texttt{ok}, h, \sigma, s \ \rightsquigarrow \ h', \sigma', \epsilon$ *and* $\epsilon \neq \texttt{castExc}$ *all hold, then* $\Gamma; \Delta' \vdash h', \sigma' \ \wedge \ \epsilon = \texttt{ok}$.

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. Furthermore, because initialisation states are not present at runtime, we need to infer the expected initialisation state for an object via the static types of references to that object. 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 both ι and the object referred to by the stack variable exist before execution, then ι was

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 particularly deserves explanation. It 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 abovementioned properties to strengthen our induction hypothesis; we prove the following lemma (from which Theorem 1 follows), which includes properties 1–9:

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

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

PROOF 1. *By (elaborate) induction on the derivation of* $\texttt{ok}, h, \sigma, s \rightsquigarrow h', \sigma', \epsilon$. *The full proof and accompanying lemmas are available in our technical report [23].*

5. Extensions

In this section we discuss how to support additional language features and approaches to make our type system even more expressive.

5.1 Concurrency

Our type system naturally extends to concurrency. The only requirement that is necessary to preserve soundness is that each object is thread-local until it is initialised. That is, we maintain an invariant that any shared object (reachable from more than one thread) is initialised. Sharing uninitialised objects could lead to cross-type aliases when the object reaches its commitment point in the thread that created it.

In Java this invariant can be maintained by two rules. Firstly, only initialised `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 can only reach initialised objects. Secondly, only committed references can be stored in static fields (see below). 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.

The rules for concurrency are an example where we use initialisation types to *prevent* escaping (namely escaping of a free reference from the creating thread) rather than making escaping safe (as for the escaping from constructors).

5.2 Arrays

Arrays do not have constructors and their number of elements might not be statically known. Therefore, it is not easily possible to use a definite assignment analysis to determine when an array has been initialised. Delayed types [11] allow programmers to call a special marker method to indicate that an array of non-null elements has been initialised; the method performs a runtime check to ensure that the array elements are indeed non-null.

We adopt this approach and use the return from the marker method as the commitment point for the array. However, since we do not store initialisation states at runtime, we cannot check at runtime that the array elements are themselves initialised, that is, we cannot check that the array is deeply initialised. Therefore, we ensure deep initialisation of the array by a type rule. An array update of the form $a\,[i]\ =\ e$ requires e to have a committed type. This rule is more restrictive than the corresponding rule for field updates; it does not allow one to store free objects in *any* arrays and, therefore, does not support the initialisation of cyclic structures that include an array of non-null elements.

The initialisation modifier of a new array with a non-null element type is unclassified. An array reference is never free, which avoids cross-type aliases when the array reaches its commitment point. The initialisation modifier of a new array with a possibly-null element type is committed. Such arrays do not have an invariant to establish and (as all arrays) can never reach uninitialised objects. So they can safely be regarded as committed at creation.

5.3 Static Fields

Static fields belong to classes rather than objects and are initialised by static class initialisers rather than constructors. It is in general not possible to determine modularly when static class initialisers execute and, thus, when a static field is initialised. Therefore, we use a conservative type rule: static fields must not have non-null types and may only be assigned null and committed objects. The latter requirement is necessary to guarantee that objects cannot reach uninitialised objects when they reach their commitment point (see Sec. 3.5) and to handle concurrency (see Sec. 5.1).

5.4 Factory Methods

One alternative approach to initialisation is the use of *factory methods*, in which complex initialisation code is performed in a (usually static) method rather than a constructor. We could extend our technique to handle factory methods, at the expense of more complex annotations. Firstly, to allow interesting initialisation in factory methods, we would need to support a special kind of "weak constructor" which is *not* obliged to initialise the fields of the new object. Weak constructors could only be invoked via new expressions in factory methods (or super calls from weak constructors), and would always return a free reference. Further, we would need the ability to mark certain methods as factory methods, which *would* be tasked with initialising the returned object (which must have been newly-allocated). Calls to factory methods could then be treated similarly to our rules for handling new expressions. So far, such an extension has not appeared to be worth the additional complexity involved.

5.5 Subclassing

Extending our formalisation with subclassing and inheritance affects the definite assignment analysis for constructors and the dataflow analysis in general Sec. 3.6. The definite assignment analysis requires each constructor to initialise the non-null fields of the enclosing class. Fields declared in superclasses will be initialised by a superclass constructor. Since our field initialisation is monotonic, this is sufficient to ensure that all non-null fields of an object have been initialised when the new-expression terminates; that is, when the last constructor of the object has run.

The dataflow analysis may safely assume that after a call to a super-constructor, each non-null field f declared in a superclass contains a non-null value. So it is safe to give `this.`f a non-null type even though `this` is free. After a call to a constructor of the same class, one may additionally assume that the non-null fields of that class have been initialised.

Note that this design avoids having to parameterise our types with type frames (as in raw types), to express partial initialisation: our free references, along with the dataflow analysis, already give us the expressiveness we require. Note further that we do not need to prevent dynamically-bound method calls on free references (a common source of initialisation errors); our type system will check that such calls can handle free receivers, which will in turn force the method implementation not to assume that the receiver is initialised.

5.6 Generics

We allow type arguments for generic classes to include non-nullity modifiers but not initialisation modifiers. The solution for non-nullity is adopted from Spec#. Parameterising a class with initialisation modifiers isn't very useful because field types cannot have initialisation modifiers. For example, a committed instance of class `List<T>` may store only committed objects; thus it would not be meaningful to instantiate the type parameter `T` with a free or unclassified type.

It is potentially useful to parameterise methods with initialisation modifiers. For instance, an identity-method works for each of the three initialisation modifiers. To avoid having to define several copies of such a method for different initialisation modifiers, we can support for polymorphism over initialisation modifiers in method signatures. We omitted the feature here and in our implementation because we have not yet seen code "in the wild" that needs the extra expressiveness provided. However, we did include it in our extended formalisation and soundness proof [23].

5.7 Committed-only Fields

The type system presented so far does not generally handle situations where a constructor stores a committed reference into a field of its receiver, and then reads it back to perform some computation on it. For example, consider the following code (based on an example we found while experimenting with our implementation):

```
class C {
  Stack! s;  // library class
  public C(object o) {
    this.s = new Stack(); // committed value
    ...                   // other code
    this.s.push(o); // fails to type-check
  }
}
```

The problem here is that the `push` method of class `Stack` does not (and cannot be expected to) support anything but a committed receiver. However, since `this` is free, the field read `this.s` has an unclassified type and, thus, the call does not type check. One might initially think that this problem can be best handled by extending the dataflow analysis to remember the initialisation states of values stored in fields.

However, one cannot (soundly and modularly) preserve this information across method calls because the method might reassign the field, possibly with a free value.

We observed that many fields are only ever used to store committed values (at all program points), and for such fields, one would prefer to make this discipline explicit and use the information to refine our type system's expressiveness. In fact, it is sufficient to distinguish two kinds of fields: *committed-only* and *standard* fields. A committed-only field may only be assigned values which have committed types. Any value which is read from a committed-only field can (if known not to be null) be assumed to refer to an initialised object; we therefore give such field-reads a committed type even when reading from a receiver which is not committed.

Committed-only fields proved useful in many practical examples. In fact, we found it most fruitful in our experiments to make committed-only the default declaration for fields, and to introduce explicit annotations only for those fields which need to store non-committed references at some point during initialisation. This means a few extra annotations are required for the initialisation of interesting cyclic structures, but on the other hand examples like the one above can be supported without annotations.

5.8 Invariants

We presented our type system as an extension to a non-null type system, but it is far more general. Our approach supports all monotonic one-state and two-state invariants that satisfy the following two requirements.

First, it must be possible to determine that the invariant holds for the new object at the end of a new-expression. For non-null types, we achieve that with a flow analysis. For other one-state invariants, one could use an assertion that the invariant holds; the assertion can then be checked at runtime or verified statically. For two-state invariants such as immutability, no check is required; it suffices to check that the invariant holds for all pairs of states from now on.

Second, it must be possible to check that the invariant is monotonic. For non-null types, we achieve that by preventing programs from storing null in non-null fields. For other one-state and two-state invariants, one could add an assertion to each field update with a committed or unclassified receiver that checks a condition that is sufficient for the preservation of the invariant (for instance in the form of update guards [3]). For immutability, this assertion would always fail.

6. Experimental Evaluation

In order to evaluate our type system in practice, we wrote a modified version of the Spec# compiler [15], implementing our type system. Starting from Spec# gave us the practical advantage that the existing non-null type checking and dataflow analysis could be reused. Our implementation adapted and replaced the implementation of delayed types; in most cases we were greatly aided by being able to adapt or extend the existing code written by Manuel Fähndrich and Songtao Xia. In the implementation (unlike our formalisation), we made use of the dataflow analysis to *infer* the initialisation states for local variables. This makes the system much more usable for substantial code. Our compiler implements the type system presented in this paper, with additional support for base calls, static fields and methods, and arrays (see Sec. 5.2). In the course of our experiments, we realised that the committed-only-fields extension (discussed in Sec. 5.7) would enable us to handle many more cases, and so we also implemented this extension.

We tested our implementation on two fairly large codebases - a version of SSCBoogie (the Spec# verifier, which is written entirely in Spec#), and an old version of the widely-used Boogie program verifier (written in Spec#; we used an older version because the Boogie project has been migrated to pure C# since June 2010). As well as these two large projects, we also tested our compiler against the Spec# collections used by the compiler itself, and by-hand encodings of the examples found in this paper. All of this code was already written with non-null annotations (but without appropriate initialisation annotations).

Our approach was to start from the code without any initialisation-related annotations, and first see how many type-checking warnings were issued by our compiler; this indicates how many cases were not already handled by the defaults in our type system. We then investigated how many of these warnings could be eliminated by the addition of [Free] and [Unclassified] annotations, without performing any other changes to the code itself. This process was very mechanical; in the end, it amounted to the systematic application of three rules:

1. When the type system warned that the receiver of a method call was expected to be initialised, but was not guaranteed to be so (i.e., its static type was free or unclassified) we annotated the signature of the called method with [Unclassified].
2. When the type system warned that an argument to a method call was expected to be initialised, but was not guaranteed to be so (i.e., its static type was free or unclassified) we annotated the formal parameter of the called method with [Unclassified].
3. When the type system warned that we attempted to store a non-committed value in a committed-only field, we removed the committed-only status from the field.

The results of this annotation effort are shown in Fig. 9. In some cases (particularly for the first two rules above), these rules had to be iterated; when we mark a new receiver of a method as [Unclassified], for example, this means that any uses of the receiver inside the method body might no longer type-check. In the case of the third rule, it could be that parts of the code already depend on the committed-only status of the field; in this case, unless those parts could themselves

	Boogie	SSCBoogie	Other
Lines of code	43996	15672	1739
Total warnings	43	108	19
Annotations used	74	28	19
Warnings removed	42	106	18
Warnings remaining	1	2	1

Figure 9. *Experimental evaluation results.* Our modified version of the Spec# compiler was run on two large projects: a Spec# version of the Boogie verifier, and the verifier for the Spec# language itself, SSCBoogie. We also ran our compiler on several small, challenging examples, including those used in this paper (included together under "Other" in the table). We show the total type-checking warnings generated for the un-annotated code, indicating how many initialisation problems are not handled by the defaults in our type system. We then annotated the code in a mechanical fashion, to see how many warnings could be removed. The (few) remaining cases indicate that some code refactoring was still needed to make the code type-check, by moving some code which depends on initialisation being complete, to outside of a constructor body.

be fixed with annotations as above, we marked the case as one which required refactoring (cf. "Warnings remaining" in the table). This also applied if we found that we needed to add annotations to code to widely-used superclasses, since these would in general prohibit interesting implementations. Similarly, we were not able to annotate any library code.

The results provide convincing evidence that our type system is usable; we found only four points in the code examined where we couldn't make the code type-check simply by adding appropriate annotations. The first of these involved calling a method on the receiver in a constructor, and then within the method relying on a non-null field containing a non-null value. Two cases involved passing the `this` reference from a constructor as an argument to an overridden method call for which we could not re-annotate the superclass signature. The final case is the example discussed in Sec. 7.2. All four cases could be handled by moving the problematic lines to outside of the constructor (which makes sense in general, since there might still be subclasses to initialise after the constructor executes).

The number of annotations required in our experiments is very low; on average about one annotation per warning about initialisation, and per about 500 lines of code. In fact, it was often the case that several warnings could be removed by a single annotation, while in the worst case we had to provide thirteen annotations to deal with one original issue, when an escaping object was passed between many calls before finally being captured in the field of another new object. Because of our positive experiences and the soundness guarantees our type system provides, we plan that our implementation will replace the current release of Spec#.

We anticipated worse results than we actually discovered, because the old approach to initialisation in Spec# supported optional "non-delayed constructors", which encourage a programming style in which extra code can be (soundly) included in the body of a constructor, after initialisation is known to be completed. These "non-delayed constructors" initialise the type-frames of an object bottom-up rather than top-down; one must initialise the non-null subclass fields before the superclass ones. This initialisation must take place *before* the "base" ("super", in Java) call is made in the constructor body. In this way, one can be sure that *after* the base call is made, the object has been initialised at all type-frames, and therefore can take part in arbitrary code. We chose not to support this feature in our type system or implementation, mainly because it requires runtime behaviour which is not typically supported by mainstream OO languages; for example, these constructors cannot be supported directly in Java or C#. We were pleasantly surprised to find that, even given an initial codebase which included many complex constructor definitions which allowed the newly constructed object to escape in interesting ways, our type system was able to handle virtually all cases easily. We judge this to be because the complicated constructors still typically enforced an informal discipline for handling escaped objects; such objects were sometimes captured in the fields of other objects under initialisation, but almost never had their own fields written to, and those fields which were read from an escapee object typically only ever stored committed values or null, at all program points. Thus, the combination of free references (particularly inside constructors), [Unclassified] annotations to support the passing of escaped objects, and committed-only fields, allows to programmer to enforce these apparent informal policies in a way which can be expressed and directly checked in our type system.

7. Related Work

7.1 Avoiding Initialisation Bugs

We first revisit some of the related work discussed in Sec. 2, with respect to a simple example of *faulty* object initialisation, shown in Fig. 10. 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 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

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

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

  public C(C! p) {
    this.setF(p);       // alias p as this.f
    this.f.setF(this);  // assign this to p.f
    this.g = p.f.g.f;   // null ptr exception
  }
}
```

Figure 10. Example of faulty object initialisation.

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.

Let's now discuss how our system prevents the faulty example from Fig. 10. 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 modifier. 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 is rejected by the type checker (the receiver of the update is not free and the right-hand side is not committed). 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.

7.2 Comparison with Masked Types

Since we believe Masked Types to be the most expressive comparable approach to object initialisation, we provide a more-detailed comparison with our approach here. Fig. 11 shows the running example from the Masked Types paper [21]. Masked Types allow a programmer to flexibly express a wide variety of different refinements of a class type. For example, while the types `Leaf` and `Binary` are standard OO class types, the type `Binary\parent!\left[root.parent]\right[root.parent]` describes a reference to a `Binary` object whose `parent` is not assigned, and whose `left` and `right` fields may refer to a objects which cannot be assumed to be fully initialised until the field `root.parent` is assigned a value (this is called a *conditional mask* on the fields). Using the various kinds of field masks included in their type system, it is possible to statically describe arbitrary combinations of uninitialised fields and mutually-dependent conditions under which masks can be lifted, and the fields read from. A programmer can potentially express precisely under which conditions a field can be soundly assumed to be permanently initialised, on a per-field basis. Furthermore, method and constructor signatures are annotated with explicit effects which describe how the mask information associated with references passed to the call evolves during the method execution. In contrast to our system, the programmer is not forced to initialise all non-null fields before a constructor terminates; instead, a constructor can employ an effects annotation to make explicit the state of each uninitialised field, potentially in terms of conditional masks which can later be lifted in the client code.

Masked Types are highly expressive; they can encode arbitrarily complicated idioms in a precise and statically-checked way. But this complexity inevitably finds its way into the type system itself, even at the source level. As we explained in our design goals, we believe that an important criterion for widespread adoption of a type system is *simplicity*, which encompasses both the conceptual understanding required to use the system, and the level of annotation required for typical programming idioms. The Masked Types syntax for annotations includes grammars for flexible effects annotations and sequenced masks, abstractions over masks, constraints on these abstract masks, and so on. The concepts and notations a programmer must learn in order to understand and use this type system are both numerous and sophisticated in nature. Furthermore, fully understanding the typing rules can be quite 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". For these reasons, we believe that the technical complexity and richness of the type annotation language, while extremely powerful, makes the system unsuitable for widespread use by programmers, which was our overall design aim.

A modified version of the binary tree example (using our type system) is shown in Fig. 12. To be able to type the example, we had to make two changes. Firstly, the original example initialises the `parent` of the new `Binary` object outside the constructor (in the last line of Fig. 11). Since our system enforces that constructors initialise all non-null fields, we added a default assignment of the `parent` field in class `Node`. Secondly, the `Binary` constructor in the origi-

```
class Node {
  Node! parent;
  Node() effect *! -> *! { }
}

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

final 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 fully initialised
```

Figure 11. Tree with back-pointers using masked types.

nal 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 considered committed in our example code. The reason the masked types typing is sound (even though it points the constructor arguments at an object under initialisation), is that the constructor signature builds in the requirement that the `l` and `r` parameters passed *must* not yet have had their `parent` fields initialised. Effectively, this means that their constructor definition can only be used with `Leaf` instances which are themselves still under initialisation. Our constructor, on the other hand, must be called with initialised `Leaf` arguments (we could weaken this requirement my annotating the parameters as unclassified); in the client code our `Node` constructor takes care of the default initialisation for us. Note that our refactored code would remain typeable without extra annotations even if the `Leaf` and `Binary` classes were not declared `final`, which is not the case for the original code.

```
class Node {
  Node! parent;
  Node() { parent = this; }
}

class Leaf extends Node { }

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

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

Figure 12. Tree with back-pointers in our type system.

The comparison of the two versions of the example illustrates that our version reduces the annotation overhead tremendously. We do not have to add a single annotation to handle the initialisation of the cyclic structure[6]. Our system does not directly 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). However, the implementation is also more general; we do not require `Leaf` objects to be partially initialised in order to add them to `Binary` objects.

We believe that these differences illustrate different motivations. Masked Types provide a very general and expressive solution (which can handle more-complex typing disciplines with regard to initialisation), and 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.

7.3 Other Related Work

Haack and Poll present a type system for object immutability [12], which has some similarities to our work. As they remark in Section 4 of their paper, the initialisation problem for immutability is simpler because one does not need

[6] Although, with the committed-only-fields extension (Sec. 5.7) we need a single annotation on the field `parent` to override our chosen default.

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 [24]. However, the work of Haack and Poll can support initialisation of immutable structures by introducing extra generic "qualifier" arguments to methods, and using these to explicitly scope and then end the initialisation phase for a group of objects. This requires extra annotation overhead, although some annotations can be inferred for their system. In recent work, Zibin et al. present a type system which combines ownership types with immutability [25]. This system can handle cyclic data structures, provided they are initialised under a single common owner object; in our terminology, the owner's commitment point can be used to implicitly define the commitment point of all owned objects, which can be initialised flexibly in the meantime.

Various implementations and practical works have been based on the original proposals of Fähndrich and Leino [10] (with their "raw types" approach to object initialisation). Several implementations have been based on pluggable types frameworks [4]. Andreae et al. developed the JavaCOP framework, and implemented a non-null type checker in the framework [2]. Ekman and Hedin [7] have written a pluggable types implementation of the type system on top of their JAstAdd compiler framework [8]. Papi et al. have developed the Checker framework [20], to facilitate the flexible development of type systems based on customisable Java annotations. This framework has since been used to develop many type checkers for different properties [9]. As future work, we aim to develop a pluggable types implementation using one of these frameworks, so that we can also evaluate our design against Java code.

Hubert et al. [14] present a machine-checked analysis for inferring non-null types, and Hubert has also extended this work to the level of Java bytecode [13]. Male et al. [16] also present a bytecode verification for non-null types, while Chalin and James [5] 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.).

Spoto and Ernst [22] have recently presented an interprocedural flow analysis (implemented in the Julia tool) for inferring "raw" annotations from unannotated Java bytecode; their technique can also be broadly applied to other initialisation-related properties.

In other recent work, Zibin et al. [26] present a type-system for object initialisation in the open-source language X10. Their work has similar design goals to ours in terms of simplicity and soundness, but they are more restrictive; they require that dynamic method calls be forbidden on objects under initialisation (for the sake of ease of understanding by the user). The implemented version of their type system also does not support cyclic patterns of initialisation, although a possible extension to handle this is sketched.

8. Future Work and Conclusions

8.1 Future Work

Since the annotation efforts required in our experiments turned out to be very mechanical, it seems natural to investigate the possibility of developing inference tools to provide or suggest annotations for existing code. We have not yet seen many cases in practice where such inference would need to be sophisticated, but in principle there are some interesting design choices to be made. For example, so far whenever we found that a method was called with a non-committed actual parameter, we tried annotating the method formal parameter with [Unclassified]. However, in cases where such a method is *only* called with free actual parameters, one might gain flexibility by choosing a [Free] annotation; this would allow the passed parameter to have its fields written to. So far in the code we have examined, we have not seen any cases where an object escapes from its constructor and then has its fields written to via other methods; it seems that this (difficult to reason about) coding pattern is typically avoided.

The experimental results obtained with our implementation are very promising, and we plan to extend our experiments to further codebases. Since Spec# is a superset of C# 2.0, one possibility is to port existing C# codebases to Spec#, allowing us to experiment with annotating widely-used class implementations. We plan to migrate our implementation to the open source version of the Spec# compiler. As mentioned above, we also plan to implement our type system in a pluggable types framework for Java; this will provide a simple way to access large bodies of critical code, and see how well our annotations work for, e.g., the Java standard libraries. It will also provide a more convenient means for other researchers to experiment with using our type system directly. We are also interested in developing prototype implementations for other suitable languages, and to apply our type system to the initialisation of invariants other than non-null types. We have been informed that the Eiffel development team plan to adopt and implement our approach to handle initialisation in the Eiffel language [18].

8.2 Conclusions

We have presented a novel type-based approach to object initialisation, based on a simple distinction between objects known to be under initialisation, and objects known to be initialised. The core of our system has been formalised and proven sound, specifically for the problem of handling object initialisation for non-null types, but in a way which generalises to other monotonic object invariants such as immutability. Our type system is implemented, and experi-

ments on large codebases have yielded promising results both in terms of expressiveness and ease of use. While our implementation is based around Spec#, the type system itself is suitable for use in any heap-based language with explicit constructors (or an equivalent language concept).

Since the goal of our work was to design a system suitable for mainstream use, let us revisit the design goals which we highlighted in the introduction.

Our type system is *modular*; each method is type-checked independently of the others, and the usual rules for co- and contra-variant method overriding allows our analysis to remain ignorant of overriding subclass implementations.

The type system is *sound*; we have provided a detailed formalisation for a small language which illustrates the critical aspects of the problem, and the full soundness proofs are available online in our technical report [23].

Our presented solution is suitably *expressive*; our experiments on large volumes of code show that the defaults in our type system handle the vast majority of constructors; almost all remaining cases can be dealt with simply by the straightforward addition of type annotations.

Last, but not least, our type system is *simple*. Understanding how to use the system requires classifying objects into just two initialisation states, and handling references using just three initialisation modifiers (the most common being the default). Our experiments show that the modifiers are rarely needed, and are generally sufficient to handle interesting initialisation patterns which do arise, both in practice and in research papers. We believe that the conceptual simplicity of our approach, along with its low annotation burden and lack of required runtime support, make it a promising candidate for future use in mainstream programming languages.

Acknowledgments

We are grateful to the anonymous referees for extensive and constructive feedback. We would like to thank Manuel Fähndrich for helpful discussions of delayed types and the previous Spec# implementation. We thank Hermann Lehner and Sophia Drossopoulou for useful discussions on the details of the formalisation and presentation. We especially thank Arsenii Rudich for many discussions on the inception and details of this work, and for last-minute food supplies.

References

[1] PMD tool. `http://pmd.sourceforge.net/`, 2002.

[2] C. Andreae, J. Noble, S. Markstrum, and T. Millstein. A framework for implementing pluggable type systems. In *OOPSLA*, pages 57–74. ACM, 2006.

[3] M. Barnett and D. Naumann. Friends need a bit more: Maintaining invariants over shared state. In *MPC*, volume 3125 of *LNCS*, pages 54–84. Springer, 2004.

[4] G. Bracha. Pluggable type systems. In *OOPSLA'04 Workshop on Revival of Dynamic Languages*, 2004.

[5] P. Chalin and P. R. James. Non-null references by default in Java: Alleviating the nullity annotation burden. In *ECOOP*, pages 227–247, 2007.

[6] ECMA. *ECMA-367: Eiffel analysis, design and programming language*. ECMA, 2006.

[7] T. Ekman and G. Hedin. Pluggable checking and inferencing of non-null types for Java. *Journal of Object Technology*, 6 (7), 2007.

[8] T. Ekman and G. Hedin. The jastadd extensible Java compiler. In *OOPSLA*, pages 1–18. ACM, 2007.

[9] M. D. Ernst and M. Ali. Building and using pluggable type systems. In *FSE*, pages 375–376. ACM, 2010.

[10] M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. In *OOPSLA*, pages 302–312. ACM, 2003.

[11] M. Fähndrich and S. Xia. Establishing object invariants with delayed types. In *OOPSLA*, pages 337–350. ACM, 2007.

[12] C. Haack and E. Poll. Type-based object immutability with flexible initialization. In *ECOOP*, LNCS, pages 520–545. Springer, 2009.

[13] L. Hubert. A non-null annotation inferencer for Java bytecode. In *PASTE*, pages 36–42. ACM, 2008.

[14] L. Hubert, T. P. Jensen, and D. Pichardie. Semantic foundations and inference of non-null annotations. In *FMOODS*, pages 132–149, 2008.

[15] K. R. M. Leino and P. Müller. Using the Spec# language, methodology, and tools to write bug-free programs. In *LASER Summer School 2007/2008*, volume 6029 of *LNCS*, pages 91–139. Springer, 2010.

[16] C. Male, D. J. Pearce, A. Potanin, and C. Dymnikov. Java bytecode verification for @NonNull types. In *CC*, LNCS, pages 229–244. Springer, 2008.

[17] B. Meyer. Attached types and their application to three open problems of object-oriented programming. In *ECOOP*, pages 1–32, 2005.

[18] B. Meyer. Personal commmunication, 2011.

[19] B. Meyer, A. Kogtenkov, and E. Stapf. Avoid a void: The eradication of null dereferencing. 2010.

[20] M. M. Papi, M. Ali, T. L. Correa Jr., J. H. Perkins, and M. D. Ernst. Practical pluggable types for Java. In *ISSTA*, pages 201–212, 2008.

[21] X. Qi and A. C. Myers. Masked types for sound object initialization. In *POPL*, pages 53–65, 2009.

[22] F. Spoto and M. D. Ernst. Inference of field initialization. In *ICSE'11*, Waikiki, Hawaii, USA, May 25–27, 2011.

[23] A. J. Summers and P. Müller. Freedom before commitment : Simple flexible initialisation for non-null types. Technical Report 716, ETH Zurich, 2011.

[24] M. S. Tschantz and M. D. Ernst. Javari: adding reference immutability to Java. In *OOPSLA*. ACM, 2005.

[25] Y. Zibin, A. Potanin, P. Li, M. Ali, and M. D. Ernst. Ownership and immutability in generic java. In *OOPSLA '10*, 2010.

[26] Y. Zibin, D. Cunningham, I. Peshansky, and V. Saraswat. Object initialization in X10. In *X10 Workshop*, 2011.