Masked Types for Sound Object Initialization

Xin Qi Andrew C. Myers

Computer Science Department
Cornell University
{qixin,andru}@cs.cornell.edu

Abstract

This paper presents a type-based solution to the long-standing problem of object initialization. Constructors, the conventional mechanism for object initialization, have semantics that are surprising to programmers and that lead to bugs. They also contribute to the problem of null-pointer exceptions, which make software less reliable. Masked types are a new type-state mechanism that explicitly tracks the initialization state of objects and prevents reading from uninitialized fields. In the resulting language, constructors are ordinary methods that operate on uninitialized objects, and no special default value (`null`) is needed in the language. Initialization of cyclic data structures is achieved with the use of conditionally masked types. Masked types are modular and compatible with data abstraction. The type system is presented in a simplified object calculus and is proved to soundly prevent reading from uninitialized fields. Masked types have been implemented as an extension to Java, in which compilation simply erases extra type information. Experience using the extended language suggests that masked types work well on real code.

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]: Class invariants, programming by contract; D.3.2 [*Language Classifications*]: Object-oriented languages

General Terms Languages, Reliability

Keywords invariants, null pointer exceptions, conditional masks, cyclic data structures, data abstraction

1. Introduction

Object initialization remains an unsatisfactory aspect of object-oriented programming. In the usual approach, objects of a given class are created and initialized only by class constructors. Therefore, when implementing class methods, the programmer can assume that object fields satisfy an invariant established by the constructors. However, in the presence of inheritance, the methods of partly initialized objects may be invoked before the invariant has been established. As a result, reasoning about object initialization can be challenging and non-modular. No fully satisfactory solution to object initialization currently exists.

This paper presents a new solution to the object initialization problem, based on a new type mechanism, *masked types*. As Section 2 describes, a masked type keeps track of the parts of an object that have not been initialized. For example, the type $T\backslash f$ describes an object of type T whose field f may not be initialized yet, and the type $T\backslash *$ represents an object none of whose fields are necessarily initialized. As an object is constructed, the type of the object changes to reflect the fields that are initialized. Thus, the type system for masked types is flow-sensitive; it has *typestate* [31]. The type of an object conservatively tracks its initialization state, so a partially initialized object cannot be used where a fully initialized object is expected.

The problem of object initialization is intertwined with the problem of null pointer exceptions, which significantly hurt software reliability [2]. Because object initialization is unsound, most languages aiming for type safety (e.g., Java, C#, Modula-3) first initialize fields with `null`. This semantics implies that `null` must be a legal value for all object types, leading to ubiquitous, implicit null checks that can generate null pointer exceptions. Recently there has been interest in controlling null pointer exceptions through *non-null annotations* and other means [7, 2, 19, 6]. Non-null annotations by themselves do not solve the problem of object initialization; in fact, they make it more important because non-null fields must be initialized before use. But with masked types, there is no need for a default initialization value. It is then straightforward to eliminate `null` values entirely from the language. There are legitimate uses of `null` other than as an initialization placeholder, but for these uses, an "option" or "maybe" type is a better approach, because it makes null checks explicit and rare.

A language with masked types can be simpler in another way. There is no need to give constructors a special status in the language, because types track initialization state. Rather than a language feature, constructors become a design pattern: they are ordinary methods that change the initialization state of the receiver.

Cyclic data structures pose a challenge for object initialization. However, *conditionally masked types* make it possible to create cyclic data structures, such as doubly-linked lists and trees with parent pointers, without resorting to placeholder `null` values. Conditional masks record dependencies between initialization of different fields, so that initializing one field can "tie the knot", changing the initialization state of many fields at once.

Perhaps the most closely related prior work is that of Fähndrich and Xia [9], who introduce *delayed types* for static reasoning about partially initialized objects. Masked types support cyclic data structures that delayed types do not. Masked types also support richer initialization abstractions: for example, helper methods for partial initialization and reinitialization of recycled objects. *Abstract masks*, described in Section 3, support initialization abstractions that are compatible with data abstraction and inheritance.

Masked types have been formalized for a simplified object language, described in Section 4. The key soundness theorem is formalized and has been proved for this language: well-typed programs never read uninitialized fields.

POPL'09, January 18–24, 2009, Savannah, Georgia, USA.

```
1  class Point {
2    int x, int y;
3    Point(int x, int y) {
4      this.x = x;
5      this.y = y;
6      display();
7    }
8    void display() {
9      System.out.println(x + " " + y);
10   }
11 }
12
13 class CPoint extends Point {
14   Color c;
15   CPoint(int x, int y, Color c) {
16     super(x, y);
17     this.c = c;
18   }
19   void display() {
20     System.out.println(x + " " + y + " " + c.name());
21   }
22 }
```

Figure 1. Code with an initialization bug

Section 5 reports on the implementation of masked types as a mostly backward-compatible extension to the Java language called J\mask. Section 6 discusses experience using J\mask in the context of the Java Collections Framework, where masked types are shown to do a good job of capturing desirable initialization idioms. Related work is discussed in Section 7. Section 8 concludes.

2. Masked types

Figure 1 illustrates a bug that can easily happen in an object-oriented language like Java. In the class `Point`, representing a 2D point, the constructor calls a virtual method `display` that prints the coordinates of the point. The two fields `x` and `y` are properly initialized before `display` is called. However, in the subclass `CPoint` representing a colored point, the `display` method has been overridden in a way that causes the added `c` field to be read before it is initialized, resulting in a null pointer exception.

This example is simple, but in general, initialization bugs are difficult to prevent in an automatic way. It would be too restrictive to rule out virtual method calls on partially constructed objects. Further, the bug involves the interaction of code from two different classes (`Point` and `CPoint`). An implementer of `CPoint` might not have access to the code of `Point` and would not realize the danger of overriding the `display` method in this seemingly reasonable way.

Our goal is to prevent code like that of class `Point` from type-checking, but to allow complex, legitimate initialization patterns. The key observation is that before the call to `display` on line 6, the fields in `Point` are initialized, but fields of subclasses of `Point` are not. However, the type of the method `display` does not prevent the partially initialized receiver from being passed to an overridden version of the method that reads uninitialized fields, as in `CPoint`.

2.1 Types for initialization state

A masked type $T\backslash M$, where M is a *mask* that denotes some object fields, is the type T but without read access to the denoted fields. Masked types are a completely static mechanism, so a J\mask program is compiled by erasing masks. No run-time penalty is paid for safe object initialization.

The simplest form of a mask is just the name of a field. For example, an object of type `CPoint\c` is an instance of the `CPoint` class whose field `c` cannot be read, perhaps because it has not been initialized. We say that the field `c` is *masked* in this type.

A type with no mask means that the object is fully initialized. In typical programming practice, this would be the ordinary state of the object, in which its invariant is already established.

On entry to a constructor such as `Point()`, the newly created object has all its fields masked. The actual class of the new object might be a subclass (for example, `CPoint`), so on exit, subclass fields remain to be initialized. A *subclass mask*, written C`.sub`, is used to mask all fields introduced in subclasses of C, not including those of C itself. Therefore, just before line 4 in Figure 1, the object being constructed has type `Point\x\y\Point.sub`. (While this type looks complicated, it can be inferred automatically.)

When a field is initialized by assigning to it, the corresponding mask is removed from the type of the object. For example, line 4 in Figure 1 assigns to field `x`, so the type of `this` becomes `Point\y\Point.sub`. After the assignment to `y` on the next line, the type of `this` becomes `Point\Point.sub`. Thus, the initialization of various fields is recorded in the changing type of `this`. Because variables may have different types at different program points, J\mask has a *flow-sensitive* type system.

Subclass masks such as `Point.sub` can be removed when the exact run-time class of an object is known, because there are no subclass fields left to initialize. The type of a `new` expression is known exactly, as is the type of a value of any class known not to have a subclass (in Java, a "final" class).

J\mask has a special mask $*$ as a convenient shorthand for masking all fields, including those masked by the subclass mask. On entry to the `CPoint` constructor, the object can be given type `CPoint\`$*$, which is equivalent to `CPoint\x\y\c\CPoint.sub`.

2.2 Mask effects

In J\mask, methods and constructors can have *effects* [23] that propagate mask information across calls. For example, the J\mask signatures for the `Point` constructor and the `display` method can be annotated explicitly with effect clauses:

```
Point(int x, int y) effect * -> Point.sub
void display() effect {} -> {}
```

The effect of this `Point` constructor says that at entry to the constructor, all fields are uninitialized (precondition mask `*`) and therefore unreadable; at the end of the constructor, only fields introduced by subclasses of `Point` remain uninitialized (postcondition mask `Point.sub`). Because the initial and final masks of the `display` method are both `{}`, denoting the absence of any mask, the method can be called only with a fully initialized object, and it leaves the object fully initialized.

With these effects, the bug in Figure 1 would be caught statically. The method `display` cannot be invoked on line 6, because there the type of `this` is `Point\Point.sub`, which does not satisfy the precondition of `display`. The J\mask compiler detects this unsafe call without inspecting any subclass of `Point`.

This example suggests how mask effects make the J\mask type system modular. Mask effects explicitly represent the contract on initialization states that a method is guaranteed to follow. This explicit contract allows the compiler to type-check programs one class at a time, and also enables programmers to reason about initialization locally.

Indeed, masked types and mask effects capture changes to initialization state with enough precision that constructors in J\mask are essentially ordinary methods that remove masks from the receiver. However, for convenience and backward compatibility, the J\mask language still has constructors.

To reduce the annotation burden, the J\mask language provides default effects for methods and constructors. Programmers do not

normally have to annotate code with effects or masks. For ordinary methods, the default is `{} -> {}`; for constructors, the default effect is close to that shown above (see Section 2.3).

The effects shown capture changes to the initialization state of the parameter `this`, the receiver object. J\mask also supports effects on other parameters, as shown in Section 2.5.

For simplicity, exceptions, which are rarely thrown during initialization anyway, have been ignored in this paper. However, exceptions can be supported by providing a postcondition for each exceptional exit path in the effect clause.

2.3 Must-masks

All the masks shown in Section 2.1 are *simple* masks. A simple mask S, e.g., f, $*$, or $C.\texttt{sub}$, means that the fields it describe *may* be uninitialized. Thus, there is a subtyping relationship $T \leq T\backslash S$, because it is safe to treat an initialized field as one that may be uninitialized.

However, when an object is created, it is known that all the fields *must* be uninitialized. J\mask uses *must-masks*, written $S!$, to describe fields that must definitely be uninitialized. A must-masked type $T\backslash S!$ is also a subtype of $T\backslash S$, but T is not a subtype of $T\backslash S!$.

One use of must-masks is for initialization of "final" fields, which is only allowed when the field is must-masked, ensuring that the field is initialized exactly once. Must-masks and the absence of masks roughly correspond to the notions of *definite unassignment* and *definite assignment* in the Java Language Specification [12]. However, J\mask ensures that a final field cannot be read before it is initialized, while Java does not. J\mask also lifts the limitation in Java that final fields can only be initialized in a constructor or an initializer.

Must-masks are also used to express the default effect of a constructor of class `C`, which is `*! -> C.sub!`. Objects start with all fields definitely uninitialized, which is represented with the initial mask `*!`. Constructors usually do not initialize fields declared in subclasses, so the default postcondition mask is `C.sub!`.

Must-masks impose restrictions on how an object can be aliased: if there is a reference with a must-masked type, it must be the only reference through which the object may be accessed; otherwise, the must-masked field might be initialized through another reference to the object, invalidating the must-mask. This does *not* preclude aliasing, but implies rather that other references have to be through fields that are themselves masked.

J\mask uses typestate to keep track of initialization state. A problem with most previous typestate mechanisms is that they require reasoning about potential aliasing, to prevent aliases to the same object that disagree about the current state. Aliasing makes it notoriously difficult to check whether clients and implementations are compliant with protocols specified with typestate [1], and much previous work on typestates requires complicated aliasing annotations or linear types. J\mask is designed to work with no extra aliasing control mechanism, which provides the added benefit of soundness in a multi-threading setting, since operations on an object through aliases from other threads do not invalidate typestates in the current thread.

The key to avoiding reasoning about aliasing is that if an assignment creates an unmasked alias, then must-masks on both sides are conservatively converted to corresponding simple ("may") masks. For example, after the following code, the type of both `x` and `y` is the simply masked type `C\f`:

```
C\f! x = ...;
C\f! y = x;
```

Similarly the following code also removes the must annotation from the type binding of variable `x`, because `z.g` becomes an alias and the field `g` is not masked in the type `D` of variable `z`:

```
C\f! x = ...;
D z = ...;
z.g = x;
```

The non-aliasing requirement on must-masks might seem restrictive, but it is usually not a problem: must-masks typically appear near allocation sites, where no alias has been created.

2.4 Reinitialization

Beyond initialization, masked types can help reasoning about *reinitialization*. A mask can represent not only an uninitialized field, but also a field that must be reassigned before further read accesses. To enforce reinitialization, a mask can be introduced on the field, via the subtyping rule $T \leq T\backslash f$.

For example, Figure 2 illustrates a custom memory management system that manages a pool of recycled objects of the class `Node`. Actively used objects are not in the pool and store data in their `d` fields. Objects in the pool are threaded into a freelist using their `next` fields. When a `Node` object is no longer used, it is put into a pool by calling the `recycle` method; when a new instance of `Node` is needed, the `getNode` method returns an object from the pool, if there is any. Masked types can help ensure that the field `d` is reinitialized whenever a `Node` object is retrieved from the pool and gets a second life. Of course, like most custom memory management systems, the code in this example does not guarantee that no alias exists after an object is recycled. Masked types are not intended to enforce this kind of general correctness.

```
1  class Node {
2    Data d;
3    Node\d next;
4  }
5
6  class Pool {
7    Node\d head;
8    ...
9    Node\d\next getNode() {
10     if (head != sentinel) {
11       Node\d\next result = head;
12       head = head.next;
13       return result;
14     } else
15       return new Node();
16   }
17   void recycle(Node\next n) {
18     n.next = head;
19     head = n;
20   }
21 }
```

Figure 2. Object recycling

The type `Node` is a subtype of `Node\d`, and therefore the second assignment (line 19) in method `recycle` type-checks, causing `Node` objects in the pool to "forget" about the data stored in field `d`.

Masked types provide an additional benefit here. Objects in active use have type `Node\next`, preventing traversal of the freelist from outside the `Pool` class.

2.5 Initializing cyclic data structures

Many data structures that arise in practice contain circular references: for example, doubly linked lists and trees whose nodes have parent pointers. Safe initialization of these cyclic data structures poses a challenge. In object-oriented languages, storing a reference to a partially initialized object is normally required, with no guarantee that the object is fully initialized before use.

J\mask explicitly tracks fields that point to partially initialized objects with *conditionally masked types*, written

$T\backslash f[x_1.g_1,\ldots,x_n.g_n]$. The conditional mask $f[x_1.g_1,\ldots,x_n.g_n]$ describes a field f referencing a partially initialized object, which will become fully initialized when all fields $x_i.g_i$ are initialized. In other words, the removal of the mask on f is conditioned on the removal of all masks on $x_i.g_i$.

Conditional masks are normally introduced by an assignment to a must-masked field f, when the right-hand side of the assignment has more masks than the declared field type. Consider, for example, a field assignment `x.f = y`, where `x` has type $T\backslash f!$, `y` has type $T'\backslash g$, and the field `f` of class T has type T'. Note that $T'\backslash g$ is not a subtype of T'. J\mask makes this assignment safe by changing the type of `x` to $T\backslash f[y.g]$ after the assignment, showing that the field `x.f` is still masked, but its mask should be removed upon the removal of the mask on `y.g`.

```
1  class Node {
2    Node parent;
3    Node() effect *! -> *! { }
4  }
5
6  final class Leaf extends Node {
7    Leaf() effect *! -> parent! { }
8  }
9
10 final class Binary extends Node {
11   Node left, right;
12   Binary(
13     Node\parent!\Node.sub[l.parent] -> *[this.parent] l,
14     Node\parent!\Node.sub[r.parent] -> *[this.parent] r)
15   effect *! -> parent!, left[this.parent],
16                right[this.parent] {
17     this.left = l;
18     this.right = r;
19     l.parent = this;
20     r.parent = this;
21   }
22 }
23
24 Leaf\parent! l = new Leaf();
25 Leaf\parent! r = new Leaf();
26 Binary\parent!\left[root.parent]\right[root.parent]
27     root = new Binary(l, r);
28 root.parent = root; // Now root has type Binary.
```

Figure 3. Initialization of a tree with parent pointers

Figure 3 shows how to safely initialize a binary tree with parent pointers. For convenience, we assume all local variables, including formal parameters, are `final`. (Section 5 discusses how to relax this.)

Figure 3 also demonstrates effects on parameters other than the receiver `this`: the parameters `l` and `r` of the `Binary` constructor both have the type `Node`$\backslash *$`[this.parent]` upon the exit of the constructor.

In this example, initialization is bottom-up, as it would be, for example, in a shift-reduce parser. Child nodes are created, initialized, and then used to construct their parent node. However, child nodes cannot be fully initialized before their `parent` fields are set, and moreover, they cannot even be considered fully initialized before the fields of all the objects that are transitively reachable are set. (Top-down initialization of this data structure creates similar issues.)

The `parent` field of a node will eventually point to an object that is created later and that contains child pointers pointing back to the current node, creating parent–child cycles. Of course, the `parent` field of the root of the tree must point to something special. For example, it can point to the root itself, as shown on line 28, or to a sentinel node.

The dependencies between masks after line 20 in Figure 3 are summarized in Figure 4, where the mask at the tail of an arrow is removed when the mask at its head is removed. The masks on `this.left` and `this.right` after line 20 transitively depend on the mask on `this.parent`.

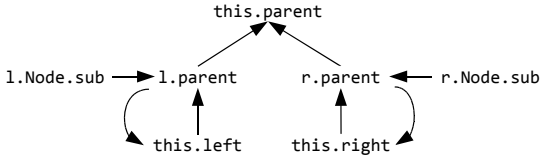

Figure 4. Mask dependencies

The postcondition in the effect of the `Binary` constructor summarizes the dependencies in the figure: parameters `l` and `r` both have mask $*$`[this.parent]`, which means that all their fields are conditionally masked, and `this` has type `Binary`$\backslash$`parent!`$\backslash$`left[this.parent]`$\backslash$`right[this.parent]`, which is compatible with the parameter type of the `Binary` constructor. Therefore, the construction can proceed to build higher trees. Finally, the tree is fully initialized when the `parent` field of the root is initialized, because removing its mask enables removing all the masks in Figure 4.

In general, a field f should be unreadable unless every object transitively reachable through f has been appropriately initialized. That is, its masks have been removed at least to the level according to the type of the field through which the object is referenced.

Therefore, there are three ways to remove a conditional mask on field f:

- Like other kinds of masks, the conditional mask can be removed by directly initializing the field f.
- As shown in Figure 3, on line 28, conditional masks on `root.left` and `root.right` are removed by removing the mask `root.parent` they (transitively) depend on.
- The last way to remove a conditional mask is by creating cyclic dependencies. For example, the following code creates cyclic dependencies between `x.f` and `y.g`, which cancel each other.

  ```
  // x starts with type C\f!, and y starts with D\g!
  x.f = y; // now x has type C\f[y.g]
  y.g = x; // now y has type D\g[x.f]
           // x can be typed C, and y can be typed D
  ```

  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.

Subtyping generalizes to conditionally masked types: $T \leq T\backslash f[x_1.g_1,\ldots,x_n.g_n] \leq T\backslash f$. In fact, a type T with unmasked field f can be viewed as a type that has empty conditions for the mask on f, that is, $T\backslash f[]$, and a simply masked type $T\backslash f$ can be seen as having an unsatisfiable condition on f, because a simple mask cannot be removed by removing other masks.

Conditional masks and simple masks do not impose any restriction on aliasing, because mask subtyping ensures that they cannot be invalidated by any future change to the object. This property has been called heap monotonicity [8].

Conditional masks also provide a way to create temporarily unreadable aliases for must-masked objects. Because the aliases are unreadable, the must annotations need not be removed. In Figure 3, for example, the assignment on line 17 creates an alias `this.left` for the left child object stored in variable `l`, but `l` remains of type `Node`$\backslash$`parent!`, since the field `this.left` is masked with the conditional mask `left[l.parent]` after line 17. Not losing the

must information means the initialization state of `l` is tracked more precisely.

For simplicity, fields currently must be declared with unmasked or simply masked types; no conditional masks or must-masks are allowed. It should be straightforward to add support for conditionally masked field types, but this is left for future work.

3. Abstract masks

With the exception of $*$ and C.`sub`, the masks we have seen so far are *concrete*, explicitly naming instance fields. Concrete masks create difficulties for data abstraction, because the fields might not be visible where the masks are needed. For example, in Figure 3, if the two fields `left` and `right` of class `Binary` were private, it would be impossible to declare the local variable `root` as shown on line 26, because its type mentions the names of the fields outside the class definition.

```
1  class Node {
2    mask Children;
3    ...
4  }
5
6  final class Binary extends Node {
7    private Node left, right;
8    mask Children += left, right;
9    Binary(...)
10   effect *! -> parent!,
11                Children[this.parent] { ... }
12   ...
13 }
14 ...
15 Binary\parent!\Children[root.parent]
16     root = new Binary(l, r);
17 root.parent = root;
```

Figure 5. The tree example with abstract masks

Therefore J\mask introduces *abstract masks* that abstract over sets of concrete fields, providing a way to write types that mask fields that are not visible. Figure 5 shows an updated version of the code from Figure 3, where the two fields `left` and `right` are now private, and an abstract mask `Children` is introduced to mask them outside the class `Binary`. The `Children` mask is first declared in class `Node` (line 2), with an empty set of fields, and is *overridden* in `Binary` (line 8) to include the two children of a binary node. J\mask currently allows abstract masks to be overridden only to include more fields; more complex overriding is left to future work.

The $*$ mask, introduced in Section 2.1, is not much different from any other abstract mask, except that it is built-in, and is automatically overridden in every class to include all the fields declared in that class.

3.1 Modular checking of abstract masks

Subclass masks. The `Point/CPoint` example in Section 2.1 showed that unsafe calls to overridden methods could be caught in a modular way with the help of the subclass mask `Point.sub`. The mask `Point.sub` can be connected to the abstract mask $*$ through the equivalence of the two types `Point\*` and `Point\x\y\Point.sub`. Any type with an abstract mask can be similarly expanded. For example, given the code in Figure 5, the masked type `Binary\Children` is equivalent to `Binary\left\right\Binary.Children.sub`, where `Binary.Children.sub` represents all the concrete masks that are added into overriding declarations of `Children` in subclasses of `Binary`, excluding `Binary` itself. The set {`left`,`right`,`Binary.Children.sub`} is the *interpretation* of `Children` in the context of `Binary`.

In general, $C.M$.`sub` represents the subclass mask of abstract mask M with respect to class C, and the interpretation of M in the context of C is a set consisting of all the concrete masks added into M in C and its superclasses, together with subclass mask $C.M$.`sub`. Before type checking, the J\mask compiler internally expands all abstract masks into their interpretations.

Subclass masks are important for modular type checking, because they make it possible to distinguish the current definition of an abstract mask and overriding definitions in subclasses, which are generally unavailable in a modular setting.

```
1  class C {
2    T f;
3    mask M += f;
4    void initM() effect M -> {} {
5      this.f = ...;
6    }
7  }
8
9  class D extends C {
10   T g;
11   mask M += g;
12   void initM() effect M -> {} {
13     this.g = ...;
14     super.initM();
15   }
16 }
```

Figure 6. Code that needs mask constraints

Mask constraints. Subclass masks help prevent unsafe calls, but since they describe fields that are generally not known in the current class, safely removing them by initialization requires some additional mechanism. Figure 6 illustrates an initialization helper method `initM`, which is intended to remove the abstract mask `M` from its receiver. It is properly overridden in the subclass `D` to handle the overridden abstract mask `M`. However, the `initM` method would not type-check as written in Figure 6, because right after line 5, the type of `this` is actually `C\C.M.sub`, rather than the unmasked type `C`.

J\mask uses *mask constraints* to solve this problem. Every J\mask method can declare a mask constraint of the form `captures` $M_1,\ldots,M_n$, where $M_1,\ldots,M_n$ are abstract masks. This constraint means that the body of the method is type-checked assuming that the masks M_i are the same as their concrete definition in the class where the method is defined, with no subclass masks.

For example, the signature of `initM` on lines 4 and 12 can be updated with a mask constraint:

```
void initM() effect M -> {} captures M
```

The example then type-checks, because at the entries to `initM` in classes `C` and `D`, the type of `this` becomes `C\f` and `D\f\g` respectively, rather than `C\f\C.M.sub` and `D\f\g\D.M.sub`.

However, when type-checking callers against the public signature of the method, the abstract mask should still be interpreted to include the subclass mask.

A method defined in class C with a mask constraint on an abstract mask M depends on the set of fields that M denotes in C. It would be unsound to allow that method to be inherited by a subclass that overrides the abstract mask. Therefore, the type system requires such methods to be overridden when the masks they depend on are overridden. Consequently, constructors, final methods, and static methods cannot have mask constraints, because they cannot be overridden in subclasses.

programs	Pr	$::= \langle \overline{L}, e \rangle$
class declarations	L	$::= \texttt{class}\ C\ \texttt{extends}\ C'\ \{\overline{F}\ \overline{Mt}\}$
field declarations	F	$::= T\ f$
method declarations	Mt	$::= T\ m(\overline{T}\ \overline{x})\ \texttt{effect}\ \overline{M_1} \rightsquigarrow \overline{M_2}\ \{e\}$
simple masks	S	$::= f \mid \mathsf{sub}_C$
masks	M	$::= S \mid S! \mid S[\overline{p}.\overline{S_p}]$
paths	p	$::= \ell \mid x$
unmasked types	U	$::= \circ \mid C \mid C!$
types	T	$::= U \mid T\backslash M$
expressions	e	$::= (T\ p) \mid \texttt{new}\ C \mid e_1; e_2 \mid e.f$
		$\mid (T_1\ p_1).f = (T_2\ p_2) \mid (T_0\ p_0).m((\overline{T}\ \overline{p}))$
		$\mid \texttt{let}\ T\ x = e_1\ \texttt{in}\ e_2$
typing environments	Γ	$::= \emptyset \mid \Gamma, x{:}T \mid \Gamma, \ell{:}T$
heaps	H	$::= \emptyset \mid H, \ell \mapsto o$
objects	o	$::= C!\backslash\overline{M}\{\overline{f} = \overline{\ell}\}$
evaluation contexts	E	$::= [\cdot] \mid E.f \mid E; e \mid \texttt{let}\ T\ x = E\ \texttt{in}\ e$

Figure 7. Grammar

3.2 Mask algebra

J\mask supports two algebraic operations on masks that make abstract masks more useful: $(M_1 + M_2)$ and $(M_1 - M_2)$.

An abstract mask can be interpreted as a set of concrete masks on fields and possibly a subclass mask. The two operators on masks correspond to the set union $(+)$ and set difference $(-)$ of the interpretations of the abstract masks. Concrete masks can appear in algebraic masks, where they are interpreted as singleton sets.

Algebraic masks enable the programmer to express initialization state abstractly, without knowing all the fields masked by an abstract mask. For example, suppose there is a local variable x, starting with the type $T\backslash M$ where M is an abstract mask, and field $x.f$ is initialized:

```
T\M x = ...;
x.f = ...;    // The type of x is now T\(M - f)
```

Here, one needs not know which concrete masks are included in M, nor even whether M includes f.

Mask algebra also helps programmers compose masks to keep the typestates in J\mask compact. For example, if a class has n fields, each of which might independently be initialized or uninitialized, it would require 2^n different typestates to represent all possible initialization states, were there no mask algebra. With mask algebra, one can simply use the "sum" of the masks corresponding to all the uninitialized fields.

J\mask currently only supports these two algebraic operations on masks, but they seem to suffice. Richer operators on masks are left to future work.

4. The J\mask calculus

We now formalize masked types as part of a simple object calculus. Unfortunately, previous object calculi are not suitable for modeling masked types.

4.1 Grammar

Figure 7 shows the grammar of the core J\mask calculus. We use the notation $\overline{a}$ for both the list $a_1, \ldots, a_n$ and the set $\{a_1, \ldots, a_n\}$, for $n \geq 0$. We abbreviate terms with list subterms in the obvious way, e.g., $\overline{T}\ \overline{x}$ stands for $T_1\ x_1, \ldots, T_n\ x_n$, $T\backslash\overline{M}$ stands for $T\backslash M_1\backslash\ldots\backslash M_n$, and $p.\overline{S}$ stands for $p.S_1, \ldots, p.S_n$.

A program Pr is a pair $\langle \overline{L}, e \rangle$ of a set of class declarations L and an expression e (the `main` method). Each class C is declared with a superclass C', a set of field declarations $\overline{F}$ and a set of method declarations $\overline{Mt}$. To simplify presentation, all the class declarations are assumed to be global information.

J\mask only supports single inheritance. The root of the class hierarchy is denoted by $\circ$. We write $C \prec C'$ to mean that class C is a direct subclass of C', and the relation $\prec^*$ is the reflexive and transitive closure of $\prec$.

Notably, there is no `null` value in the language, because none is needed for object initialization.

There are three kinds of masks: simple masks S, must-masks $S!$, and conditional masks $S[\overline{p}.\overline{S_p}]$. The auxiliary function simple elides the must annotation and conditions of a mask.

$$\mathsf{simple}(S) = S$$
$$\mathsf{simple}(S!) = S$$
$$\mathsf{simple}(S[\overline{p}.\overline{S_p}]) = S$$

There are two kinds of simple masks: concrete field masks f, and subclass masks sub_C, that is, `C.sub` in the J\mask language. The calculus does not explicitly model the abstract mask `*`, because it can be expanded into a collection of field masks and a subclass mask. For the simplicity of the semantics, other abstract masks and mask constraints are omitted.

We require that in a well-formed type, no two masks mention the same field, and every variable appearing in a condition is in the domain of the typing environment. The order of masks in a type does not matter, so $T\backslash f_1\backslash f_2 = T\backslash f_2\backslash f_1$.

An unmasked type U is either a normal class type C or an *exact* class type $C!$. An object of $C!$ must be an instance of class C, and not of any proper subclass of C. (This overloads the "!" symbol, which is also used for must-masks.) The source of exactly typed values is object creation, because the expression `new` C has type $C!$. Exact types are useful because they make removal of subclass masks possible, as discussed in Section 2.1.

An object is created with expression `new` C, which adds a fresh memory location to the heap, with all fields uninitialized. Uninitialized fields are not represented in the heap, so there is no need for `null`. Initialization is done by calling appropriate methods.

To simplify presentation of the semantics and the proof of soundness, we allow only paths p (local variables x at compile time, or heap locations ℓ at run time) to appear in field assignments and method calls. This does not restrict expressiveness, because of `let` expressions.

Every read through a path p is represented as an expression $(T\ p)$, where the annotation T is a statically known type. The annotation is primarily to make the proof of soundness easier; in the actual J\mask implementation, T is inferred by the compiler.

Typing environments Γ contain type bindings for both variables x and heap locations ℓ. Bindings for locations are extracted from the heap and are used to type-check expressions during evaluation.

The J\mask calculus models the heap as a function from memory locations l to objects o. The formalization attaches a type to every object on the heap, in addition to value bindings for the fields. The object type is always based on some exact class type, which is known at run time. The type might also have masks, and since the base class is always exact, no subclass mask may appear on the heap. Masks in the operational semantics are included only for the soundness proof and can be erased in the implementation.

4.2 Class member lookup

Figure 8 shows auxiliary functions for looking up class members. For a class C, $\mathsf{ownFields}(C)$ and $\mathsf{ownMethods}(C)$ are the set of fields and methods declared in C itself, and $\mathsf{fields}(C)$ and $\mathsf{methods}(C)$ also collect those declared in all the superclasses of C. $\mathsf{fnames}(\overline{F})$ is the set of all the field names in field declarations $\overline{F}$. For simplicity, we assume no two fields have the same name.

$\boxed{\Gamma \vdash T \le T'}$

$$\Gamma \vdash T \le T \ \text{(S-REFL)} \qquad \frac{\Gamma \vdash T_1 \le T_2 \quad \Gamma \vdash T_2 \le T_3}{\Gamma \vdash T_1 \le T_3}\ \text{(S-TRANS)} \qquad \frac{\vdash C \prec C'}{\Gamma \vdash C \le C'}\ \text{(S-SUP)} \qquad \Gamma \vdash C! \le C\ \text{(S-EXACT)}$$

$$\frac{\Gamma \vdash T_1 \le T_2}{\Gamma \vdash T_1 \backslash M \le T_2 \backslash M}\ \text{(S-MASK)} \qquad \Gamma \vdash T \backslash S[] \approx T\ \text{(S-EMPTY-COND)} \qquad \Gamma \vdash T \backslash S[\overline{p.S_p}] \le T \backslash S[\overline{p.S_p}, p'.S']\ \text{(S-COND-SUB)}$$

$$\frac{S = \mathsf{simple}(M)}{\Gamma \vdash T \backslash M \le T \backslash S}\ \text{(S-SIMPLE)} \qquad \frac{\mathsf{sub}_C = \mathsf{simple}(M)}{\Gamma \vdash C! \backslash M \approx C!}\ \text{(S-EXACT-MASK)} \qquad \frac{p' : C! \backslash \overline{M} \in \Gamma}{\Gamma \vdash T \backslash S[\overline{p.S_p}, p'.\mathsf{sub}_C] \approx T \backslash S[\overline{p.S_p}]}\ \text{(S-EXACT-COND)}$$

$$\frac{\vdash C \prec C' \quad \mathsf{fnames}(\mathsf{ownFields}(C)) = \overline{f} \quad \mathsf{sub}_{C'} = \mathsf{simple}(M)}{\Gamma \vdash T \backslash M \approx T \backslash \mathsf{expand}(M, \{\overline{f}, \mathsf{sub}_C\})}\ \text{(S-SUBMASK)} \qquad \frac{\vdash C \prec C' \quad \mathsf{fnames}(\mathsf{ownFields}(C)) = \overline{f}}{\Gamma \vdash T \backslash M[p.\mathsf{sub}_{C'}, \overline{p'.S}] \approx T \backslash M[p.\overline{f}, p.\mathsf{sub}_C, \overline{p'.S}]}\ \text{(S-SUBMASK-COND)}$$

$\boxed{\Gamma \vdash p : T}$

$$\frac{p : T \in \Gamma}{\Gamma \vdash p : T}\ \text{(TP-PATH)} \qquad \frac{\Gamma \vdash \ell : T_1 \quad \Gamma \vdash T_1 \le T_2}{\Gamma \vdash \ell : T_2}\ \text{(TP-SUB)} \qquad \frac{\Gamma \vdash p : T \backslash f[p.f, \overline{p'.S}]}{\Gamma \vdash p : T \backslash f[\overline{p'.S}]}\ \text{(TP-COND-CYCLE)}$$

$$\frac{\Gamma \vdash p : T \backslash S[p'.f, \overline{p''.S'}] \quad \Gamma \vdash p' : T' \quad f \notin \mathsf{masked}(T')}{\Gamma \vdash p : T \backslash S[\overline{p''.S''}]}\ \text{(TP-COND-ELIM)} \qquad \frac{\Gamma \vdash p : T \backslash S[p'.S', \overline{p''.S''}] \quad \Gamma \vdash p' : T' \backslash S'[\overline{p'''.S'''}]}{\Gamma \vdash p : T \backslash S[\overline{p''.S''}, \overline{p'''.S'''}]}\ \text{(TP-COND-TRANS)}$$

$\boxed{\Gamma \vdash_R e : T, \Gamma'}$

$$\frac{\Gamma \vdash x : T \quad x : T_x \in \Gamma \quad \Gamma' = \Gamma\{\!\{x : \mathsf{noMust}(T_x)\}\!\} \quad \Gamma' \vdash \mathsf{noMust}(T) \le T'}{\Gamma \vdash_R (T\ x) : T', \Gamma'}\ \text{(TR-VAR)} \qquad \frac{\Gamma \vdash \ell : T \quad \Gamma \vdash T \le T' \quad \ell : T_\ell \in \Gamma \quad \Gamma' = \Gamma\{\!\{\ell : \mathsf{noMust}(T_\ell)\}\!\}}{\Gamma \vdash_R (T\ \ell) : T', \Gamma'}\ \text{(TR-LOC)}$$

$$\frac{\Gamma \vdash e_1 : T_1, \Gamma_1 \quad \Gamma_1 \vdash_R e_2 : T_2, \Gamma_2}{\Gamma \vdash_R e_1;\ e_2 : T_2, \Gamma_2}\ \text{(TR-SEQ)} \qquad \frac{\Gamma \vdash e : T, \Gamma' \quad e \ne (T\ x) \wedge e \ne e_1;\ e_2}{\Gamma \vdash_R e : T, \Gamma'}\ \text{(TR-OTHER)}$$

$\boxed{\Gamma \vdash e : T, \Gamma'}$

$$\frac{\Gamma \vdash e : T_1, \Gamma' \quad \Gamma \vdash T_1 \le T_2}{\Gamma \vdash e : T_2, \Gamma'}\ \text{(T-SUB)} \qquad \frac{\Gamma \vdash p : T}{\Gamma \vdash (T\ p) : T, \Gamma}\ \text{(T-PATH)} \qquad \frac{\Gamma \vdash e_1 : T_1, \Gamma_1 \quad \Gamma_1 \vdash e_2 : T_2, \Gamma_2}{\Gamma \vdash e_1;\ e_2 : T_2, \Gamma_2}\ \text{(T-SEQ)} \qquad \frac{\overline{f} = \mathsf{fnames}(\mathsf{fields}(C))}{\Gamma \vdash \mathtt{new}\ C : C! \backslash \overline{f}!, \Gamma}\ \text{(T-NEW)}$$

$$\frac{\Gamma \vdash_R e_1 : T, \Gamma_1 \quad x \notin \mathtt{dom}(\Gamma_1) \quad \Gamma_1, x : T \vdash e_2 : T_2, \Gamma_2 \quad \Gamma_2 = \Gamma_2', x : T' \quad \Gamma_2'' = \mathsf{remove}(\Gamma_2', x)}{\Gamma \vdash \mathtt{let}\ T\ x = e_1\ \mathtt{in}\ e_2 : T_2, \Gamma_2''}\ \text{(T-LET)}$$

$$\frac{\Gamma \vdash e : T, \Gamma' \quad T_f = \mathsf{ftype}(T, f)}{\Gamma \vdash e.f : T_f, \Gamma'}\ \text{(T-GET)} \qquad \frac{\Gamma \vdash (T_1\ p_1) : T_1, \Gamma \quad T_1 \ne T_1' \backslash f! \quad \Gamma \vdash_R (T_2\ p_2) : \mathsf{ftype}(\mathsf{grant}(T_1, f), f), \Gamma' \quad p_1 : T \in \Gamma' \quad \Gamma'' = \Gamma'\{\!\{p_1 : \mathsf{grant}(T, f)\}\!\}}{\Gamma \vdash (T_1\ p_1).f = (T_2\ p_2) : \circ \backslash \mathsf{sub}_\circ, \Gamma''}\ \text{(T-SET)}$$

$$\frac{\Gamma \vdash (T_1 \backslash f!\ p_1) : T_1 \backslash f!, \Gamma \quad \Gamma \vdash (T_2\ p_2) : T_2, \Gamma \quad T_2 = U_2 \backslash \overline{M} \quad \mathsf{ftype}(T_1, f) = U_f \backslash \overline{S_f} \quad \Gamma \vdash U_2 \le U_f \quad \overline{S} = \{S \mid S \in \mathsf{simple}(\overline{M}) \wedge (S! \in \overline{M} \vee S \notin \overline{S_f})\} \quad p_1 : T \backslash f! \in \Gamma \quad \Gamma' = \Gamma\{\!\{p_1 : T \backslash f[p_2.\overline{S}]\}\!\}}{\Gamma \vdash (T_1 \backslash f!\ p_1).f = (T_2\ p_2) : \circ \backslash \mathsf{sub}_\circ, \Gamma'}\ \text{(T-SET-COND)}$$

$$\frac{\Gamma \vdash (T_0\ p_0) : T_0, \Gamma \quad T_0 = U \backslash \overline{M} \quad p_0 : U_0 \backslash \overline{M'} \in \Gamma \quad \mathsf{mbody}(T_0, m) = T'_{n+1}\ m(\overline{T'}\ \overline{x})\ \mathtt{effect}\ \overline{M_1} \rightsquigarrow \overline{M_2}\ \{e\} \quad \Gamma \vdash T_0 \le U \backslash \overline{M_1}\{p_0/\mathtt{this}\}\{\overline{p}/\overline{x}\} \quad \forall i \in 1..n+1.\ T_i'' = T_i'\{p_0/\mathtt{this}\}\{\overline{p}/\overline{x}\} \quad \forall i \in 1..n.\ \Gamma \vdash (T_i\ p_i) : T_i'', \Gamma \quad \forall i \in 0..n.\ T_i = T''' \backslash S! \Rightarrow (T_i'' = T''' \backslash S! \wedge \forall j \ne i.\ p_i \ne p_j) \quad \Gamma' = \Gamma\{\!\{p_0 : \mathsf{update}(p_0, \overline{M'}, U_0 \backslash \overline{M_2}\{p_0/\mathtt{this}\}\{\overline{p}/\overline{x}\})\}\!\}}{\Gamma \vdash (T_0\ p_0).m((\overline{T}\ \overline{p})) : T'_{n+1}, \Gamma'}\ \text{(T-CALL)}$$

Figure 9. Static semantics

$$\frac{\mathtt{class}\ C\ \mathtt{extends}\ C'\ \{\overline{F}\ \overline{Mt}\}}{\mathsf{ownFields}(C) = \overline{F} \quad \mathsf{ownMethods}(C) = \overline{Mt}}$$

$$\mathsf{fields}(C) = \bigcup_{C' \,:\, C \prec^* C'} \mathsf{ownFields}(C')$$

$$\mathsf{methods}(C) = \bigcup_{C' \,:\, C \prec^* C'} \mathsf{ownMethods}(C')$$

$$\frac{\overline{F} = \overline{U}\ \overline{f}}{\mathsf{fnames}(\overline{F}) = \overline{f}}$$

Figure 8. Class member lookup

4.3 Subtyping

Subtyping rules are defined in Figure 9. The judgment $\Gamma \vdash T_1 \le T_2$ states that type T_1 is a subtype of T_2 in context Γ. The judgment $\Gamma \vdash T_1 \approx T_2$ is sugar for the pair of judgments $\Gamma \vdash T_1 \le T_2$ and $\Gamma \vdash T_2 \le T_1$.

Most subtyping rules are intuitive. S-COND-SUB states that adding conditions makes a conditional mask more conservative. S-SIMPLE states that a type with a must-mask or a conditional mask is a subtype of the corresponding simply masked type.

The subtyping rule S-SUBMASK uses an auxiliary function expand, which expands a mask S into a set of masks $\overline{S'}$, while preserving any annotation on S:

$$\mathsf{expand}(S, \overline{S'}) = \overline{S'}$$

$$\mathsf{expand}(S!, \overline{S'}) = \overline{S'}!$$

$$\mathsf{expand}(S[\overline{p.S_p}], \overline{S'}) = \overline{S'}[\overline{p.S_p}]$$

As shown in Figure 9, there are often a number of different ways of writing equivalent types. The five type equivalence rules (S-EMPTY-COND, S-EXACT-MASK, S-EXACT-COND, S-SUBMASK, and S-SUBMASK-COND) can be read as normaliza-

$$\mathsf{masked}(U) = \emptyset \qquad \mathsf{class}(C) = C$$
$$\mathsf{masked}(T\backslash S!) = \mathsf{masked}(T\backslash S) \qquad \mathsf{class}(C!) = C$$
$$\mathsf{masked}(T\backslash S[\overline{p}.\overline{S_p}]) = \mathsf{masked}(T\backslash S) \qquad \mathsf{class}(T\backslash M) = \mathsf{class}(T)$$
$$\mathsf{masked}(T\backslash f) = \{f\} \cup \mathsf{masked}(T)$$
$$\mathsf{masked}(T\backslash \mathsf{sub}_C) = \mathsf{masked}(T)$$

$$\frac{C = \mathsf{class}(T) \quad f \notin \mathsf{masked}(T) \quad \mathsf{fields}(C) = \overline{F} \quad F_i = T_f\ f}{\mathsf{ftype}(T, f) = T_f}$$

$$\frac{C = \mathsf{class}(T) \quad C \prec C' \quad Mt = \ldots\ m(\ldots)\ \ldots \quad \left(\begin{array}{l} Mt \in \mathsf{ownMethods}(C) \vee \\ Mt \notin \mathsf{ownMethods}(C) \wedge \mathsf{mbody}(C', m) = Mt \end{array}\right)}{\mathsf{mbody}(T, m) = Mt}$$

$$\mathsf{noMust}(U) = U$$
$$\mathsf{noMust}(T\backslash M) = \begin{cases} \mathsf{noMust}(T)\backslash S & \text{if } M = S! \\ \mathsf{noMust}(T)\backslash M & \text{otherwise} \end{cases}$$

$$\mathsf{grant}(T, f) = \begin{cases} T' & \text{if } T = T'\backslash f \\ T' & \text{if } T = T'\backslash f[\overline{p}.\overline{S}] \\ T' & \text{if } T = T'\backslash f! \\ T & \text{otherwise} \end{cases}$$

$$\mathsf{remove}(\emptyset, x) = \emptyset$$
$$\mathsf{remove}((\Gamma, p\!:\!T), x) = \mathsf{remove}(\Gamma, x), p\!:\!\mathsf{remove}(T, x)$$
$$\mathsf{remove}(U, x) = U$$
$$\mathsf{remove}(T\backslash S[x.S_x, \ldots], x) = \mathsf{remove}(T, x)\backslash S$$

$$\mathsf{update}(x, \overline{M}, T) = T$$
$$\mathsf{update}(\ell, \overline{M}, U) = U$$
$$\mathsf{update}(\ell, \overline{M}, T\backslash M') = \begin{cases} \mathsf{update}(\ell, \overline{M}, T)\backslash M' & \text{if } M_i = \mathsf{simple}(M')! \\ \mathsf{update}(\ell, \overline{M}, T)\backslash M_i & \text{if } \mathsf{simple}(M_i) = \mathsf{simple}(M') \\ \mathsf{update}(\ell, \overline{M}, T) & \text{otherwise} \end{cases}$$

Figure 10. Auxiliary definitions

tion rules, where the types on the left-hand side of $\approx$ are reduced to those on the right-hand side. Note that in each of the five rules, the type on the right-hand side is either syntactically simpler than that on the left-hand side, or converts an occurrence of a class on the left-hand side to its subclass. This ensures type normalization terminates. Normalized types have the following characteristics:

- A type $C\backslash\overline{M}$ has at most one subclass mask, which must be sub_C. A type $C!\backslash\overline{M}$ has no subclass mask.
- The condition $p.\mathsf{sub}_C$ does not show up if the path p has an exact type.
- Conditional masks have non-empty conditions.

For convenience of presentation, from now on, types are assumed to be in normal form, unless otherwise noted.

4.4 Expression typing

In the J\mask language, the evaluation of an expression might update some type bindings. For example, initializing a field removes the mask on that field, if there is one. Therefore, typing judgments, shown in Figure 9, are of the form $\Gamma \vdash e : T, \Gamma'$, where Γ' is the typing environment after evaluating e. We write $\Gamma\{\!\{p : T\}\!\}$ for environment Γ with the type binding of p updated to T.

There are two other kinds of judgments in Figure 9. The judgment $\Gamma \vdash p : T$ types a path p without updating the typing environment. The subsumption rule TP-SUB is limited to locations l, not any variables x, to ensure that the expression $(T\ x)$ has the most precise type annotation T (see T-PATH and TR-VAR). The judgment $\Gamma \vdash_R e : T, \Gamma'$ is used in T-LET and T-SET for typing the right-hand side of assignment, and in M-OK for typing the return expression (see Section 4.5). It avoids creating aliases for variables with type bindings that have must-masks. However, aliases are allowed if they are created with conditional masks, as shown in T-SET-COND, where no TR- rule is used.

Figure 10 defines auxiliary functions used in the typing rules. Most of them are self-explanatory. The function update, used in T-CALL, updates the type binding of the receiver according to the effect, and ensures monotonicity if the receiver is a location.

J\mask has several expression well-formedness rules, written $\vdash e$ `wf`, shown in Figure 11. The important rule is LET-WF, which imposes two requirements on `let` expressions:

- A `let` expression cannot end with a variable bound outside the scope of the `let`. For example, one cannot write `let` $T\ x = e_1$ `in` $(e_2; y)$ where y is free in the `let` expression, but rather the equivalent expression $(\texttt{let}\ T\ x = e_1\ \texttt{in}\ e_2); y$. This helps simplify type-checking of right-hand sides of assignments ($\Gamma \vdash_R e : T, \Gamma'$), so that a separate TR-LET is not necessary.
- If the variable x is bound to a location already in the scope of the `let` expression, the declared type of x cannot have any must-mask. This prevents x from being an alias with must-masks.

The expression well-formedness rules help simplify the proof of the substitution lemma (Lemma 4.5), without limiting the expressiveness of the calculus.

$$\frac{\vdash e_1\ \texttt{wf} \quad \vdash e_2\ \texttt{wf} \quad \forall x' \in \mathsf{FV}(\texttt{let}\ T\ x = e_1\ \texttt{in}\ e_2).\ e_2 \neq x' \wedge e_2 \neq e';\ x' \quad ((e_1 = (T_\ell\ \ell) \vee e_1 = e''; (T_\ell\ \ell)) \wedge \ell \in \mathsf{locs}(e_2)) \Rightarrow T \neq T'\backslash S!}{\vdash \texttt{let}\ T\ x = e_1\ \texttt{in}\ e_2\ \texttt{wf}}\ \text{(LET-WF)}$$

$$\frac{\vdash e_1\ \texttt{wf} \quad \vdash e_2\ \texttt{wf}}{\vdash e_1; e_2\ \texttt{wf}}\ \text{(SEQ-WF)} \qquad \frac{\vdash e\ \texttt{wf}}{\vdash e.f\ \texttt{wf}}\ \text{(GET-WF)}$$

$$\frac{e \neq \texttt{let}\ T\ x = e_1\ \texttt{in}\ e_2 \quad e \neq e_1; e_2 \quad e \neq e'.f}{\vdash e\ \texttt{wf}}\ \text{(OTHER-WF)}$$

Figure 11. Well-formed expressions

4.5 Program typing

Figure 12 shows the rules for checking the well-formedness of field and method declarations in a class C.

For a field declaration, the declared type may not use must-masks or conditional masks.

For a method declaration, the special variable `this` is assumed to have the precondition masks $\overline{M_1}$ at the entry point of the method, and it must be typable with the postcondition masks $\overline{M_2}$ when the method exits. Method parameters other than the receiver should remain typable with the same types at the entry. J\mask permits effects on other parameters, but for simplicity, the calculus does not support this feature. M-OK also specifies some constraints on the method effect: it cannot introduce must-masks, which is only allowed with the `new` expression; a mask in the precondition that is not a must-mask can only be replaced with a corresponding mask that is more conservative.

$$\frac{T = U\backslash\overline{S}}{C \vdash T\ f\ \mathsf{ok}} \quad \text{(F-OK)}$$

$$\frac{\begin{array}{c}\vdash e\ \mathtt{wf} \quad \Gamma = \mathtt{this}{:}C\backslash\overline{M_1},\overline{x}{:}\overline{T} \quad \Gamma \vdash_R e : T_r, \Gamma_r \\ \Gamma_r \vdash \mathtt{this}{:}C\backslash\overline{M_2} \quad \Gamma_r \vdash \overline{x}{:}\overline{T} \\ S! \in \overline{M_2} \Rightarrow S! \in \overline{M_1} \\ \left(\begin{array}{l} M \in \overline{M_1} \wedge M' \in \overline{M_2} \wedge M \neq S! \\ \wedge \mathsf{simple}(M) = \mathsf{simple}(M') \end{array}\right) \Rightarrow \vdash C\backslash M \leq C\backslash M'\end{array}}{C \vdash T_r\ m(\overline{T}\ \overline{x})\ \mathtt{effect}\ \overline{M_1} \rightsquigarrow \overline{M_2}\ \{e\}\ \mathsf{ok}} \quad \text{(M-OK)}$$

Figure 12. Program typing

4.6 Decidability of type checking

The type system of J\mask is decidable:

- For T-SUB and TP-SUB, we disallow the use of reflexivity of subtyping, and require all the rules about type equivalence ($\approx$) to be used in the direction of normalization (see Section 4.3).
- The three rules TP-COND-CYCLE, TP-COND-ELIM, and TP-COND-TRANS actually characterize a graph-theoretic reachability problem on the dependency graph (such as in Figure 4), which can be solved with depth-first search.

All other rules are syntax-directed. Therefore, type checking is decidable for J\mask.

4.7 Operational semantics

Figure 13 shows the judgments for the small-step operational semantics of J\mask, where $e,H \longrightarrow e',H'$ means that expression e and heap H step to expression e' and heap H'.

Most of the rules in Figure 13 are standard, and the notable ones are those for field assignments (R-SET and R-SET-COND), which are similar to the corresponding expression typing rules (T-SET and T-SET-COND).

In the operational semantics and in the soundness proof, typing environments are extracted from the heap, represented as $\lfloor H \rfloor$:

$$\lfloor \emptyset \rfloor = \emptyset$$
$$\lfloor H, \ell \mapsto T\ \{\overline{f} = \overline{\ell}\} \rfloor = \lfloor H \rfloor, \ell : T$$

The notation $H\{\!\{\ell := o\}\!\}$ means that the value binding of ℓ in the heap H is updated to another object o.

Figure 14 shows the heap typing rules. A heap H is well-formed, written $\vdash H$, if every field that is not masked in its container's type is bound to a location, and that location can be given a type compatible with the declared type of the field.

In H-LOC, $H(\ell, f)$ refers to the value binding of the field f of the object stored in $H(\ell)$.

4.8 Type safety

The soundness theorem of the J\mask calculus states that if an expression e is well-typed, and it can reduce to a value $(T_\ell\ \ell)$, then $(T_\ell\ \ell)$ has the same type as e. A corollary of this theorem is that object initialization is sound in the sense used elsewhere in the paper: if a program tried to read an uninitialized field, the evaluation would get stuck according to R-GET.

THEOREM 4.1. *(Soundness) If* $\vdash e$ `wf`*, and* $\vdash e : T$*, and* $e,\emptyset \rightarrow^* (T_\ell\ \ell), H$*, then* $\lfloor H \rfloor \vdash (T_\ell\ \ell) : T$.

The proof uses the standard technique of proving subject reduction and progress [35].

LEMMA 4.2. *(Subject reduction) If* $\vdash e$ `wf`*, and* $\vdash H$*, and* $\lfloor H \rfloor \vdash e : T, \Gamma$*, and* $e,H \longrightarrow e',H'$*, then* $\vdash e'$ `wf`*, and* $\vdash H'$*, and* $\lfloor H' \rfloor \vdash e' : T, \Gamma'$*, and* Γ' *is an extension of* Γ.

$\boxed{e,H \longrightarrow e',H'}$

$$\frac{e,H \longrightarrow e',H'}{E[e],H \longrightarrow E[e'],H'} \quad \text{(R-CONG)}$$

$$\mathtt{let}\ T\ x = (T_\ell\ \ell)\ \mathtt{in}\ e, H \longrightarrow e\{\ell/x\}, H \quad \text{(R-LET)}$$

$$\frac{H(\ell) = T\ \{\overline{f} = \overline{\ell}\} \quad T_i = \mathsf{ftype}(T, f_i)}{(T_\ell\ \ell).f_i, H \longrightarrow (T_i\ \ell_i), H} \quad \text{(R-GET)}$$

$$\frac{\begin{array}{c}H(\ell) = T\ \{\overline{f} = \overline{\ell}\} \quad T_\ell \neq T'\backslash f! \\ H' = H\{\!\{\ell := \mathsf{grant}(T,f)\ \{\ldots, f = \ell'\}\}\!\}\end{array}}{(T_\ell\ \ell).f = (T'_\ell\ \ell'), H \longrightarrow (\circ\backslash\mathsf{sub}_\circ\ \ell'), H'} \quad \text{(R-SET)}$$

$$\frac{\begin{array}{c}H(\ell) = T\backslash f!\ \{\overline{f} = \overline{\ell}\} \quad \mathsf{ftype}(T,f) = U_f\backslash\overline{S_f} \\ \overline{S} = \{S | S \in \mathsf{simple}(\overline{M}) \wedge (S! \in \overline{M} \vee S \notin \overline{S_f})\} \\ H' = H\{\!\{\ell := T\backslash f[\ell'.\overline{S}]\ \{\ldots, f = \ell'\}\}\!\}\end{array}}{(T_\ell\backslash f!\ \ell).f = (U\backslash\overline{M}\ \ell'), H \longrightarrow (\circ\backslash\mathsf{sub}_\circ\ \ell'), H'} \quad \text{(R-SET-COND)}$$

$$\frac{\mathsf{mbody}(T_0, m) = T_r\ m(\overline{T_x}\ \overline{x})\ \ldots\ \{e\}}{(T_0\ \ell_0).m(\overline{(T\ \ell)}), H \longrightarrow e\{\ell_0/\mathtt{this}\}\{\overline{\ell}/\overline{x}\}, H} \quad \text{(R-CALL)}$$

$$\frac{\begin{array}{c}\ell \notin \mathrm{dom}(H) \quad \mathsf{fnames}(\mathsf{fields}(C)) = \overline{f} \\ H' = H, \ell \mapsto C!\backslash\overline{f}!\{\}\end{array}}{\mathtt{new}\ C, H \longrightarrow (C!\backslash\overline{f}!\ \ell), H'} \quad \text{(R-ALLOC)}$$

$$(T\ \ell);\ e, H \longrightarrow e, H \quad \text{(R-SEQ)}$$

Figure 13. Small-step operational semantics

$$\frac{\begin{array}{c}\ell : C!\backslash\overline{M} \in \lfloor H \rfloor \quad \overline{f} = \mathsf{fnames}(\mathsf{fields}(C)) \quad \lfloor H \rfloor \vdash \ell : T \\ \forall f \in \overline{f}.\ \left(\begin{array}{l} f \notin \mathsf{masked}(T) \Rightarrow \\ H(\ell, f) = \ell' \wedge \lfloor H \rfloor \vdash \ell' : \mathsf{ftype}(T, f)\end{array}\right)\end{array}}{H \vdash \ell} \quad \text{(H-LOC)}$$

$$\frac{\forall \ell \in \mathrm{dom}(H).\ H \vdash \ell}{\vdash H} \quad \text{(HEAP-WF)}$$

Figure 14. Well-formed heaps

LEMMA 4.3. *(Progress) If* $\vdash H$*, and* $\lfloor H \rfloor \vdash e : T$ *then either* $e = (T_\ell\ \ell)$ *or there is an expression* e' *and a heap* H' *such that* $e,H \longrightarrow e',H'$.

Progress is proved by structural induction on e. To prove subject reduction, we need some preliminary lemmas.

Lemma 4.4 characterizes *extensions* of typing environments. A typing environment Γ' is an extension of Γ if:

- For every type binding $x : T \in \Gamma$, there is $x : T \in \Gamma'$;
- For every type binding $\ell : T \in \Gamma$, there is $\ell : T' \in \Gamma'$ and $\Gamma' \vdash T' \leq T$.

LEMMA 4.4. *If* Γ_2 *is an extension of* Γ_1*, and* $\Gamma_1 \vdash e : T, \Gamma'_1$*, then* $\Gamma_2 \vdash e : T, \Gamma'_2$*, and* Γ'_2 *is an extension of* Γ'_1.

PROOF: By induction on the derivation of $\Gamma_1 \vdash e : T, \Gamma'_1$. □

Lemma 4.5 shows that substituting a location for a variable preserves typing. It is used in the proof of Lemma 4.2 for method calls and `let` expressions. Before stating the substitution lemma, we first define substitution for typing environments:

An environment Γ' is the result of substituting a location ℓ of type T for a variable x in Γ, written $\Gamma' = \Gamma\{\!\{\ell/x;\ \ell : T\}\!\}$, if $\Gamma = \Gamma'', \ell : T_\ell, x : T_x$, and $\Gamma' = \Gamma''\{\ell/x\}, \ell : T$, and $\Gamma' \vdash \ell : T_\ell\{\ell/x\}$, and $\Gamma' \vdash \ell : T_x\{\ell/x\}$.

LEMMA 4.5. *If* $\Gamma = \Gamma', \ell : T_\ell, x : T_x$, *and* $\Gamma \vdash e : T, \Gamma_r$, *and* $T_\ell \neq T' \backslash S!$, *and* $T_x \neq T' \backslash S!$ *when* $\ell \in \mathsf{locs}(e)$, *then* $\Gamma\{\!\{\ell/x;\ \ell : T'_\ell\}\!\} \vdash e\{\ell/x\} : T\{\ell/x\}, \Gamma_r\{\!\{\ell/x;\ \ell : T''_\ell\}\!\}$ *for some* T''_ℓ.

PROOF: By induction on the derivation of $\Gamma \vdash e : T, \Gamma_r$. □

With these lemmas, we prove subject reduction by an induction on the derivation of $\lfloor H \rfloor \vdash e : T, \Gamma$. Then soundness (Theorem 4.1) follows directly. The proofs appear in the companion technical report [27].

5. Implementation

We have implemented a prototype compiler of J\mask as an extension in the Polyglot framework [26]. The extension code has about 3,700 lines of code, excluding blank lines and comments.

J\mask is implemented as a translation to Java. The translation is mostly by erasure, that is, by erasing all the masks, effects, and mask constraints from the code.

The compiler also applies several transformations to the J\mask source code, before erasing masks. Default effects are inserted for constructors and methods that do not have them already. To simplify type checking, initialization code, including initializers, constructors, and new expressions, is also transformed.

J\mask requires that in a conditionally masked type $T \backslash f[\overline{x}.\overline{g}]$, every x_i, including `this`, is a final local variable. However, the compiler uses a simple analysis to automatically insert the `final` modifier for local variables that are assigned only once, and for formal parameters that are never reassigned.

5.1 Inserting default effects

For a constructor of class `C`, the default effect is `*! -> C.sub!`, which describes the behavior of most constructors. The constructor starts with all the fields uninitialized, and it initializes all the fields inherited from superclasses of `C`—by calling the super constructor—and the fields declared by `C`, leaving the fields in subclasses of `C` uninitialized.

The default effect for a virtual method is `{} -> {}` because virtual methods normally work on fully initialized objects.

In our experience with using J\mask (see Section 6), these default effects work well. Programmers only have to annotate code that uses interesting initialization patterns.

5.2 Transforming initialization code

Java field declarations can include initialization expressions that are implicitly called from constructors in the same order that they appear in the class body. The J\mask compiler collects all these initializers and inserts them directly in constructors, right after super constructor calls. This initializer code is type-checked in the same way as any other constructor code.

A constructor in J\mask is just an initialization method that is called after an object is allocated on the heap. The J\mask compiler converts every constructor in the source code to a final method with the same name as the class. The transformed constructor can then be type-checked just as any other method. The compiler also inserts an empty default constructor in the generated Java code.

Every new expression `new C(...)` is split into a call to the empty default constructor to allocate the memory on the heap, and then a call to the initialization method generated from the corresponding constructor, as shown in the following piece of code:

```
final C!\(* - C.sub)! temp = new C();
temp.C(...);
```

Then the fresh local variable `temp` replaces the original expression.

5.3 Type checking

Flow sensitivity in the J\mask type system shows up only on masks, and not on any of the classes appearing in masked types. Therefore, each method is type-checked in two phases. The first phase is just normal Java type checking of the erased method code; the second phase, built upon the dataflow analysis framework provided in Polyglot, is flow-sensitive, and uses the result of the first phase as its starting point.

Once type checking is complete, masks are erased to generate Java code. This works because resolution of method overloading does not depend on parameter masks.

5.4 Inner classes

A (nonstatic) inner class is a class that is nested in the body of another class and contains an implicit reference to an instance (the *outer instance*) of the enclosing class. Every constructor of an inner class has an implicit formal parameter for the outer instance. J\mask assumes that the type of the outer instance has no masks, that is, the outer instance has been fully initialized before an instance of the inner class is created. If an inner class with a partially initialized outer instance is really needed, a transformation as described in [15] can be applied to make the outer instance explicit. J\mask currently does not directly support local classes and anonymous classes, which are inner classes nested in method bodies, although these could be converted to normal inner classes.

6. Experience

The language was evaluated by porting several classes in the Java Collection Framework (Java SDK version 1.4.2) to J\mask. The ported classes are `ArrayList`, `HashMap`, `LinkedList`, `TreeMap`, and `Vector`, together with all the classes and interfaces that they depend on. There are in total 29 source files, comprising 18,000 lines of J\mask code (exclusive of empty lines and comments).

Porting these classes to J\mask was not difficult. It was completed by one of the authors within a couple of days, including time to debug the compiler. Only 11 constructors and methods required annotation with effects or mask constraints, thanks to the default effects provided by the compiler (Section 5.1). Besides effects and mask constraints, only 11 other masked types were needed, a very small number compared to the size of the code.

The port of this code eliminated all `null`s used as placeholders for initialization. However, some `null`s were not removed:

- Java allows storing the `null` value into collections and maps.
- Some method parameters and local variables can be intentionally set to `null`, indicating that they are not available.

Among the classes we ported, the following three exhibited nontrivial initialization patterns:

6.1 LinkedList

The `LinkedList` class implements a doubly-linked cyclic list. When an instance of `LinkedList` is constructed, a sentinel node, which is an instance of the nested class `Entry`, needs to be created with its `previous` and `next` fields both pointing to itself.

The Java code first constructs an instance of `Entry` with its `previous` and `next` fields set to `null`, and then initializes the two fields with the header node itself. The following code is extracted from the constructor of `LinkedList`, where `header` is the field pointing to the sentinel node:

```
header = new Entry(null, null, null);
header.previous = header.next = header;
```

With masked types, the two fields cannot be read before they are initialized. In the constructor of the ported `LinkedList` class, the field `header` is initialized as follows:

```
header = createHeader();
```

The method `createHeader` is shown below:

```
private static Entry createHeader() {
  Entry\(* - Entry.sub)! h = new Entry();
  h.element = dummyElement;
  h.next = h;
  h.previous = h;
  return h;
}
```

The static field `dummyElement` points to an object of `java.lang.Object` because the header node does not store any real data element. Therefore, there is no need to use `null`.

6.2 HashMap

The `HashMap` class has an empty method `init`, which, according to comments in the source code, is an "initialization hook for subclasses". When a subclass of `HashMap` is created, it should override the `init` method to initialize any new subclass fields, but Java has no way to enforce this. With effects and mask constraints, the J\mask version of `HashMap` can explicitly express the contract in the signature of the method `init`:

```
void init() effect HashMap.sub -> {} captures *
```

6.3 TreeMap

`TreeMap` implements a map as a red-black tree where elements are sorted according to their keys. Each node in the tree contains fields for the left and right children, and a field pointing to its parent. A method `buildFromSorted` is used to build the tree from the bottom up, similarly to the example shown in Figure 3. Masked types support sound initialization of `TreeMap` nodes without using `null`.

6.4 Summary

Our experience is that J\mask is expressive, since it was easy to port classes with the various initialization patterns found in the Java Collection Framework. The explicit annotations in the ported code are infrequent and seem easy to understand, suggesting masked types are a natural way for programmers to enforce proper initialization of objects.

7. Related work

Non-null types. The importance of distinguishing non-null references from possibly-null references at the type level has long been recognized. Many languages, including CLU [21], Theta [22], Moby [11], Eiffel [16], ML [24], and Haskell [17], support some form of non-null and possibly-null types in their type system. In the context of Java, several proposals [2, 19, 6] have been made to support non-null types.

With non-null types, sound object initialization is usually accomplished by severely restricting expressiveness. Most existing languages with non-null types restrict how objects can be initialized; for example, some require all (non-null) fields to be initialized at once [11, 22]. This means fields and methods of an object under construction cannot be used. Further, cyclic data structures are impossible to initialize without using a placeholder value such as `null`.

Masked types are different from non-null types: when a field is masked, it is potentially uninitialized and unreadable, and therefore reading that field is statically disallowed; with non-null types, a field is always accessible regardless of how it is declared.

Fähndrich and Leino [7] make use of *raw types* to represent objects that are in the middle of being constructed, that is, objects with some non-null fields containing nulls. Methods can be declared to expect raw objects, and therefore can be called from within the constructors. Delayed types [9], extended from [7], provide a solution to the problem of safely initializing cyclic data structures, by introducing labels on object types, which represent the time by which an object is fully initialized. Delay times are associated with scopes, and form a stack at run time. Objects created with a delay time remain raw until execution exits the corresponding scope. Initialization of cyclic structures is supported by giving objects the same delay, and they become initialized together at once.

Compared to raw types, masked types provide a finer-grained representation of objects under construction. Conditional masks and delayed types are both means to track dependencies between objects under construction. However, delay times are an indirect way to represent dependencies, whereas conditional masks capture dependencies directly and explicitly. Moreover, the fact that delay times must form a stack restricts the expressiveness of delayed types in initializing cyclic structures. For example, trees where nodes have parent pointers cannot be built from the bottom up with delayed types, because one cannot coordinate the delay times of child nodes. Masked types, on the other hand, easily support this pattern, as shown in Figure 3. Masked types also have richer subtyping relationships, which can be used to enforce reinitialization.

Typestates. In most object-oriented programming languages, an object has the same type for its entire lifetime. However, objects often evolve over time, that is, having different states at different times. Typestates [31] abstractly describe object states, and when an object is updated, its typestate may also change.

Typestates have been used to express and verify various protocols [31, 4, 5, 1, 10]. Typestates have been interpreted as abstract states in finite state machines and as predicates over objects.

Masked types are not intended for checking general protocols, but rather just focus on safe object initialization. However, masks cannot be easily encoded in terms of previous typestate mechanisms. Algebraic masks, for instance, provide compact representations of partial initialization states without requiring abstract states potentially exponential in the number of fields. Conditional masks represent dependencies generated at use sites, rather than being fixed at declaration sites of predicates. Mask subtyping enriches the state space, and previous work on typestates does not appear to have anything like it.

J\mask uses subclass masks and mask constraints to ensure modular type checking. These techniques are related to rest typestates and sliding methods in Fugue [5]. However, Fugue requires that sliding methods are overridden in every subclass, whereas mask constraints in J\mask force methods to be overridden only when their watched abstract masks are overridden.

Aliasing has always been a hard problem for any typestate mechanism: first, it is not easy to maintain correct typestate information in the presence of aliasing; second, although there are typing mechanisms like linear types that help keep track of aliases, they are inconvenient for ordinary programmers. Previous work on typestates has proposed various treatments to the aliasing problem: Nil [31] completely rules out aliasing; Vault [4] and Fugue disallow further state changes once an object becomes aliased unless the changes are temporary; Bierhoff and Aldrich [1] refine the two aliasing annotations "not aliased" and "maybe aliased" in Fugue to a richer set of permissions; Fähndrich and Leino [8] also identify a kind of typestates that are heap-monotonic and work without aliasing information; Fink et al. [10] conduct whole-program verification and rely on a global alias analysis. The treatment of the aliasing problem in J\mask is inspired by [8]: simple masks and conditional masks are heap-monotonic, and must-masks, though not

heap-monotonic, are associated with newly created objects whose aliasing information is easy to track. We believe J\mask achieves a good trade-off between expressiveness and simplicity for the aliasing problem in the context of object initialization.

Masked types are reminiscent of type-based access control mechanisms that statically restrict access to individual fields or methods, e.g., [18, 28]. However, masked types are very different; they are designed for reasoning about initialization, and access is "granted" by the act of assignment to the resource, which makes little sense as an access control feature.

Static analysis. J\mask, similar to other typestate mechanisms, has a flow-sensitive type system, which can be viewed as a dataflow analysis. An alternative to masked types is an interprocedural def-use analysis, but this would lose many of the advantages of masked types. Java already has an intraprocedural analysis [32] to ensure that every local variable is definitely assigned before it is used. However, Java cannot safely prevent reading from uninitialized fields. There has been work on interprocedural def-use analysis in the context of object-oriented languages [30, 29], with varying cost and precision. This prior work detects initialization bugs on fields, but requires non-modular whole-program def-use analyses and is subject to the typically limited accuracy of whole-program alias/points-to analyses. By contrast, type checking in J\mask is modular and therefore scalable. Masked types bring another benefit because they specify the initialization contracts of methods, helping programmers reason about the code. Explicitly capturing this aspect of programmer intent seems valuable.

FindBugs [13] contains an analysis [14] that is designed specifically to detect null-pointer bugs. The analysis is neither sound nor complete, but focuses on improving accuracy. The basic analysis is interprocedural, but extensions are proposed in which non-null annotations are inserted into method signatures to represent contracts.

Shape analyses are aimed at extracting heap invariants that describe the "shape" of recursive data structures [34]. Conditional masks capture some part of the shape information of the data structure under construction. However, conditional masks are not concerned with initialized fields, and also are more about dependencies than the shape of references, and therefore have transitivity and cycle cancellation. Shape analyses are normally built upon alias analyses, and contain explicit representation of heap locations, neither of which is present in the J\mask language. J\mask only tracks mask changes on local variables, which gives it a flavor of local reasoning somewhat similar to the analysis in [3].

Because they summarize a set of concrete fields, abstract masks have some similarity to data groups [20], a mechanism used for modular program verification. Data groups do not have the equivalent of mask algebra. Moreover, masked types are about more than just abstracting fields; must-masks and conditional masks are new mechanisms that enable sound initialization of complicated data structures.

Other kinds of languages. The initialization problem is not unique to object-oriented languages. In a purely functional programming style, values are constructed all at once, avoiding the creation of partially initialized values. However, functional languages typically do not easily support the construction of cyclic data structures well, though it can be achieved in some cases with *value recursion* [33]. The typed assembly language in [25] supports initialization flags that are similar to the simple masks in J\mask.

8. Conclusions and future work

This paper introduces masked types, implemented in the language J\mask, as a solution to the problem of object initialization. Masked types provide a strong safety guarantee for initialization: uninitialized fields are never read. Further, masked types are expressive enough to support many useful initialization idioms, including objects with cyclic references. Methods and constructors in the J\mask languages explicitly express their initialization contracts through effects, which enable modular type checking, rather than requiring an expensive whole-program analysis. Because default annotations are very effective, and J\mask requires little reasoning about aliasing, J\mask has a low annotation burden. This could make the language more accessible to average programmers. Finally, by placing object initialization on a sound footing, we believe masked types can also enable other language mechanisms.

Acknowledgments

We would like to thank Sigmund Cherem, Steve Chong, Michael Clarkson, Jed Liu, and Ruijie Wang for helpful feedback on early drafts of this paper, and Doug Lea, Wojciech Moczydlowski, and Nate Nystrom for discussions. Thanks also to Jonathan Aldrich and the POPL reviewers for useful comments and suggestions.

This work was supported by National Science Foundation grants 0430161, 0627649, and CCF-0424422 (TRUST), and by the Air Force Research Laboratory, under contract #FA8750-08-2-0079. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of these organizations or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

[1] Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *Proc. 22nd ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 301–320, October 2007.

[2] Patrice Chalin and Perry James. Non-null references by default in Java: Alleviating the nullity annotation burden. In *Proceedings of the 21st European Conference on Object-Oriented Programming*, 2007.

[3] Sigmund Cherem and Radu Rugina. Maintaining doubly-linked list invariants in shape analysis with local reasoning. In *Verification, Model Checking, and Abstract Interpretation, 8th International Conference (VMCAI 2007)*, Nice, France, January 2007.

[4] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proc. SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 59–69, June 2001.

[5] Robert DeLine and Manuel Fähndrich. Typestates for objects. In *Proceedings of 18th European Conference on Object-Oriented Programming (ECOOP'04)*, 2004.

[6] Torbjörn Ekman and Görel Hedin. Pluggable checking and inferencing of non-null types for java. *Journal of Object Technology*, 6(9):455–475, October 2007.

[7] Manuel Fähndrich and K. Rustan M. Leino. Declaring and checking non-null types in an object-oriented language. In *Proc. 2003 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOSPLA)*, pages 302–312, October 2003.

[8] Manuel Fähndrich and K. Rustan M. Leino. Heap monotonic typestate. In *Proceedings of the first International Workshop on Alias Confinement and Ownership (IWACO)*, July 2003.

[9] Manuel Fähndrich and Songtao Xia. Establishing object invariants with delayed types. In *Proc. 22nd ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, October 2007.

[10] Stephen Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 133–144, 2006.

[11] Kathleen Fischer and John Reppy. The design of a class mechanism for Moby. In *Proc. SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pages 37–49, 1999.

[12] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison Wesley, 3rd edition, 2005. ISBN 0321246780.

[13] David Hovemeyer and William Pugh. Finding bugs is easy. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 132–136, 2004.

[14] David Hovemeyer, Jaime Spacco, and William Pugh. Evaluating and tuning a static analysis to find null pointer bugs. In *PASTE '05: Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 13–19, 2005.

[15] Atsushi Igarashi and Benjamin C. Pierce. On inner classes. In *Informal Proceedings of the Seventh International Workshop on Foundations of Object-Oriented Languages (FOOL 7)*, Boston, MA, January 2000.

[16] ECMA International. Eiffel analysis, design and programming language. ECMA Standard 367, June 2005.

[17] Haskell 98: A non-strict, purely functional language, February 1999. Available at `http://www.haskell.org/onlinereport/`.

[18] Anita K. Jones and Barbara Liskov. A language extension for expressing constraints on data access. *Comm. of the ACM*, 21(5):358–367, May 1978.

[19] *JSR 308: Annotations on Java Types*. Available at `http://groups.csail.mit.edu/pag/jsr308/`.

[20] K. Rustan M. Leino. Data groups: specifying the modification of extended state. In *Proc. 13th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 144–153, 1998.

[21] B. Liskov and J. Guttag. Data abstraction. In *Abstraction and Specification in Program Development*, chapter 4, pages 56–98. MIT Press and McGraw Hill, 1986.

[22] Barbara Liskov, Dorothy Curtis, Mark Day, Sanjay Ghemawat, Robert Gruber, Paul Johnson, and Andrew C. Myers. *Theta Reference Manual*. Programming Methodology Group Memo 88, MIT Laboratory for Computer Science, Cambridge, MA, February 1994. Available at `http://www.pmg.lcs.mit.edu/papers/thetaref/`.

[23] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proc. 15th ACM Symp. on Principles of Programming Languages (POPL)*, pages 47–57, 1988.

[24] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.

[25] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, May 1999.

[26] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In *Proc. 12th International Compiler Construction Conference (CC'03)*, pages 138–152, April 2003. LNCS 2622.

[27] Xin Qi and Andrew C. Myers. Masked types. Technical report, Computer and Information Science, Cornell University, October 2008. `http://hdl.handle.net/1813/11563`.

[28] Joel Richardson, Peter Schwarz, and Luis-Felipe Cabrera. CACL: Efficient fine-grained protection for objects. In *Proc. 1992 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 154–165, Vancouver, BC, Canada, October 1992.

[29] Amie L. Souter and Lori L. Pollock. The construction of contextual def-use associations for object-oriented systems. *IEEE Trans. Softw. Eng.*, 29(11):1005–1018, 2003.

[30] Amie L. Souter, Lori L. Pollock, and Dixie Hisley. Inter-class def-use analysis with partial class representations. In *PASTE '99: Proceedings of the 1999 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 47–56, 1999.

[31] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering (TSE)*, 12(1):157–171, January 1986.

[32] Sun Microsystems. *Java Language Specification*, version 1.0 beta edition, October 1995. Available at `ftp://ftp.javasoft.com/docs/javaspec.ps.zip`.

[33] Don Syme. Initializing mutually referential abstract objects: The value recursion challenge. *Electronic Notes in Theoretical Computer Science*, 148(2):3–25, 2006.

[34] Reinhard Wilhelm, Shmuel Sagiv, and Thomas W. Reps. Shape analysis. In *Proc. 9th International Compiler Construction Conference (CC'00)*, pages 1–17, 2000.

[35] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.