Establishing Object Invariants with Delayed Types

Manuel Fähndrich Songtao Xia
Microsoft Research
{maf,sxia}@microsoft.com

Abstract

Mainstream object-oriented languages such as C# and Java provide an initialization model for objects that does not guarantee programmer controlled initialization of fields. Instead, all fields are initialized to default values (0 for scalars and **null** for non-scalars) on allocation. This is in stark contrast to functional languages, where all parts of an allocation are initialized to programmer-provided values. These choices have a direct impact on two main issues: 1) the prevalence of **null** in object oriented languages (and its general absence in functional languages), and 2) the ability to initialize circular data structures. This paper explores connections between these differing approaches and proposes a fresh look at initialization. Delayed types are introduced to express and formalize prevalent initialization patterns in object-oriented languages.

Categories and Subject Descriptors F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Invariants; F.3.3 [*Studies of Program Constructs*]: Object-oriented constructs, Type structure; D.2.4 [*Software/Program Verification*]: Class invariants; D.1.5 [*Object-oriented Programming*]; D.3.3 [*Language Constructs and Features*]: Classes and objects

General Terms Languages, Reliability, Verification

Keywords non-null types, object invariants, initialization

1. Introduction

In functional languages, data structures are built bottom-up, and all parts of an allocation are initialized under programmer control. This bottom-up approach provides strong guarantees in terms of initialization: there are no initialization "holes" present that have to be filled later by programmers. A direct consequence is the absence of a **null** value in these languages, as it is not needed.

In object-oriented languages, initialization happens top-down instead. Allocation of an object default-initializes all fields to 0-equivalent values (0 for scalars, **null** for non-scalars). An object constructor is then invoked that selectively initializes some of the fields again, possibly allocating sub-objects and calling constructors on them. A direct consequence of this approach is the need to have 0-equivalent values for all types. For reference types, this means that a **null** value is part of every type. Initialization via constructors does not usually guarantee that certain fields are initialized to something other than their default value.

Circular structures Besides the initialization guarantees and the need for **null**, the top-down vs. bottom-up initialization also impacts the ability to initialize circular data structures. In object-oriented languages, it is common to build highly connected and circular structures. The top-down approach makes this easy, as a self-reference is available during the initialization of an object. This self-reference can be passed to methods and stored into fields of other objects.

In functional style bottom-up initialization, circular data structures are not directly expressible. However, some languages, such as OCaml and lazy languages, such as Haskell, do provide recursive bindings. In OCaml [8], these recursive value bindings are very restricted, as the compiler must guarantee proper initialization:

```
type T = A of T * T | B of T * T
let rec x = A( x, y )
    and y = B( y, x )
```

OCaml restricts the right-hand side of recursive value definitions to be constructors or tuples, and all occurrences of the defined names must appear only as constructor or tuple arguments. These restrictions allow the following implementation for such recursive bindings: first allocate memory for the values being constructed. In the example, this means allocating memory to hold the two triples (a tag and two values) and bind these memory locations to x and y. Once all bindings are established, the constructed values can be initialized normally. If we write out these steps in pseudo-code, they look as follows:

OOPLSA'07 October 21–25, Montréal, Québec, Canada.

Reprinted from OOPLSA'07, Proceedings of the 22nd ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, October 21–25, Montréal, Québec, Canada., pp. 1–14.

```
1  public class List {
2    Node sentinel;
3    public List() { this.sentinel = new Node(this); }
4
5    public void Insert(object data) {
6      this.sentinel.InsertAfter(data);
7    }
8  }
9
10 internal class Node {
11   List parent; Node prev; Node next;
12   object data;
13   // for sentinel construction
14   internal Node(List parent) {
15     this.parent = parent;
16     this.prev = this;
17     this.next = this;
18   }
19   // for data node construction
20   private Node(Node prev, Node next, object data) {
21     this.parent = prev.parent;
22     this.prev = prev;
23     this.next = next;
24     this.data = data;
25   }
26   internal void InsertAfter(object data) {
27     Node newNode = new Node(this, this.next, data);
28     this.next.prev = newNode;
29     this.next = newNode;
30   }
31   internal object Remove() {
32     this.next.prev = this.prev;
33     this.prev.next = this.next;
34     this.next = this;
35     this.prev = this;
36     return this.data;
37   }
38 }
```

Listing 1. Linked list example

```
let x = alloc[3];
and y = alloc[3]
in
  initACons(x, x, y);   // x.tag = A; x.0 := x; x.1 := y
  initBCons(y, y, x);   // y.tag = B; y.0 := y; y.1 := x
```

We assume the automatically generated functions initACons and initBCons that take the uninitialized block and set the tag and the two arguments. Let's now see how a similar initialization is expressed in object-oriented style. The example in Listing 1 shows the core of a doubly-linked list implementation in C# using a sentinel node and back-pointers from each node to the list. The initialization of List and Node is non-symmetric. We perform the initialization top-down, by first allocating a List object, passing the self-reference to the constructor of Node and finally assigning the sentinel to the List.sentinel field. Note that the List constructor builds an empty list, as the sentinel node is always part of a list.

```
1  class List {
2    Node! sentinel;
3    List() { this.sentinel = new Node(this); }
4
5    void Insert(object data) {
6      this.sentinel.InsertAfter(data);
7    }
8  }
9
10 class Node {
11   List! parent; Node! prev; Node! next;
12   object data;
13   // for sentinel construction
14   Node(List! parent) {
15     this.parent = parent;
16     this.prev = this;
17     this.next = this;
18   }
19   // for data node construction
20   Node(Node! prev, Node! next, object data) {
21     this.parent = prev.parent;
22     this.prev = prev;
23     this.next = next;
24     this.data = data;
25   }
26   ...
27 }
```

Listing 2. Example with non-null types

1.1 The Problem

We are interested in providing object-oriented languages with stronger invariants to express design decisions and catch errors early. In previous work [4], we proposed non-null types for object-oriented languages to deal with the prevalence of null-dereference errors. In that work, we proposed that for each reference type T, there are two versions: T? and T!, where the former includes the **null** value, whereas the latter excludes it. In the rest of this paper, we do not explicitly use the form T? and instead just write T for the possibly null type.

If we allow object fields to be declared with with non-null types, we recover the benefit of functional languages where data fields are guaranteed to be initialized and non-null. Modifying our previous example, we would like to mark the fields of List and Node as non-null. Such field invariants are crucial in checking that the store to this.next.prev on line 28 cannot cause a null-dereference.

Listing 2 shows the same code with non-null type specifications for all the fields and the necessary parameters (we omit access modifiers such as private and public from now on to reduce unnecessary details).

Fields with non-null types represent simple object invariants, guaranteeing that the corresponding field is never observably null. Due to the top-down approach of initialization in object-oriented languages however, this invariant does not necessarily hold during initialization. In particular, in the example, when the self-reference to the List object is passed to the Node constructor on line 3, the invariant of the List object is not yet established: **this**.sentinel is still **null**.

In [4], we addressed the initialization problem by

1. Forcing constructors to initialize all non-null fields
2. Introducing *raw types* to express the fact that self-references during initialization are not yet fully initialized. In constructors of class C, the self-reference **this** is thus typed as C^{raw}. Reading a field from a reference of type C^{raw} always produces a possibly null value, even if the field is declared non-null.

With raw types, we would have to annotate the List parameter on line 14 as raw, yielding the following Node constructor.

```
10 class Node {
11   List ! parent; Node! prev; Node! next;
12   object data;
13   // for sentinel construction
14   Node(List ! raw parent) {
15     this.parent = parent;   // error: List ! raw ⋠ List!
16     this.prev = this;       // error: Node!raw ⋠ Node!
17     this.next = this;       // error: Node!raw ⋠ Node!
18   }
19 ...
```

Unfortunately, this approach does not help when uninitialized object references are to be stored into fields of other objects as is the case here, unless we mark the field types as raw. We'd like to avoid using raw types for fields, as they would weaken the object invariant considerably. Thus, without changing the field types, we cannot type our example because in the contexts of the constructors, the self-reference **this** now has type List ! raw (resp. Node!raw). As a result, the assignments on lines 15–17 are ill-typed, because the field types are not raw types.

Although the raw-type system is too weak to type it, we can intuitively argue the correctness of the example: the example is correct, because the code is not relying on the field invariants until both objects are fully initialized. At that point in time, the references need no longer be viewed with raw types.

1.2 Delayed Objects

In this paper, we formalize the intuition for why the example in the previous section is correct. The principal idea is to separate allocation from initialization and to force the allocation of an object o to specify a future time t, at which o becomes valid (meaning that its field invariants hold). We refer to this time t as $o.\mathsf{ValidTime}$ in our discussion, but it is not needed at runtime. Thus, an allocation of o establishes a proof obligation that by time $o.\mathsf{ValidTime}$, all invariants of o (in particular non-nullness of fields) are established.

The passing of time is modeled as an equivalence class Now of time labels that are no longer in the future. When time t arrives, it is added to the equivalence class Now. Thus, a time $t \notin \mathsf{Now}$ is a future time, which we will write as $t > \mathsf{Now}$.

We say that an object o is delayed, if $o.\mathsf{ValidTime} > \mathsf{Now}$. Reading a field from a delayed object may yield **null**, even if the field is declared with a non-null type. The rest of the paper makes these ideas precise in the form of a static type system based on the following novel ideas:

Delayed types A delayed type $C^{\diamond t}$ describes a delayed object o with $o.\mathsf{ValidTime} = t$. Delayed types are similar to raw types in that field invariants may not yet hold. However, delayed types differ from raw types in that we can reason about *when* the referenced object becomes valid. Raw types intuitively are delayed types with an unknown delay time: $C^{\mathsf{raw}} = \exists t.C^{\diamond t}$.

Delayed fields Given an object o with delayed type $C^{\diamond t}$, we not only know that $o.\mathsf{ValidTime} = t$, we can also reason about the delay of objects q reachable from o via one or more field reads ($o \rightsquigarrow q$). In order for o to be valid at time $o.\mathsf{ValidTime}$, all such objects q must be valid by that time as well, i.e., $q.\mathsf{ValidTime} \leq t$.

This fact is convenient, as it enables storing a reference of delayed type into a field *whose type is not delayed*, as long as the target object's delay is no shorter than the stored object. As a consequence, we do not need to declare fields with delayed types, as their delay is constrained by the delay of the container.

A field assignment $o.f = q$ is thus allowed whenever $q.\mathsf{ValidTime} \leq o.\mathsf{ValidTime}$. Our type system enforces this property and thereby guarantees that invariants are not observed prematurely.

Delay scopes We separate allocation from initialization and introduce a delay scope around those two parts. Given a future time t, allocation is expressed as

```
C^◇t x = alloc C[t];
```

The newly allocated object has $x.\mathsf{ValidTime} = t$ and is assumed to be 0-initialized (**null** for non-scalar fields). The time t specifies when the invariants of the allocated object must hold. Note that there is no construtor call directly associated with an allocation.

A *delay scope* introduces a binding of a future time t that arrives at the end of the block:

```
delay t { B }
```

At the end of the delay scope, types $C^{\diamond t}$ become equivalent to $C^{\diamond \mathsf{Now}}$, which we just write as type C. In other words, at the end of the scope, objects with delay t can now be considered fully initialized.

method declaration	$\Omega(m) = \sigma_r\ C.m[\vec{t},\Gamma](y_0 : C!^{\diamond\chi_0}, y_1 : \sigma_1, .., y_n : \sigma_n)\ E_r$
field declaration	$\Omega(f) = C.f : \tau$
method definition	$\Omega(C.m) = \sigma_r\ C.m[\vec{t},\Gamma](y_0 : C!^{\diamond\chi_0}, y_1 : \sigma_1, .., y_n : \sigma_n)\ E_r\{\ e\ \}$
inheritance	$\Omega \vdash C \leq D$

Figure 1. Class and Method Declarations

Variables	$y \in V$	Classes	B, C, D
Expressions	$e ::= y \mid e.f \mid y.f := e \mid e; e$	Methods	$m \in M$
	$\mid$ **let** $y = e$ **in** e	Fields	$f \in F$
	$\mid\ y_0.m[\vec{\chi}](y_1, .., y_n)$	Effects	$E \subseteq V \times F$
	$\mid$ **ifnull** y **then** e **else** e	Types	$\tau ::= C \mid C! \mid \mathsf{void}$
	$\mid$ **delay** t **in** e	Delay types	$\sigma ::= \tau^{\diamond\chi} \mid \exists(t \leq \chi).\sigma$
	$\mid$ **let** $y =$ **alloc** $C[t]$ **in** e	Time vars	$t \in T$
	$\mid$ **unpack** $t, y = e$ **in** e	Time	$\chi ::= \mathsf{Now} \mid t$
Environment	$\Gamma ::= \bullet \mid \Gamma, y{:}\,\sigma \mid \Gamma, t$	Substitution	$\varphi \subseteq T \xrightarrow{\text{fin}} \chi$
	$\mid\ \Gamma,\ t{\leq}\chi \mid \Gamma,\ t{>}\mathsf{Now}$		

Figure 2. Language Definitions

For this to be sound, we must prove that at the end of a delay scope that binds t, all objects with delay t satisfy their invariants. For non-null field invariants, this is simple to prove: it suffices, for example, to separately ensure that block B contains calls to constructors on all allocations of time t, and to prove that each constructor initializes all non-null fields.

A strength of our technique is that it works equally for the initialization of circular object graphs as for DAGs.

1.3 Outline

The remainder of the paper is organized as follows: Section 2 introduces a small object oriented calculus and type language to formalize our approach. Section 3 describes the type rules for the language and Section 4 argues the correctness of our approach. Sections 5 and 6 describe our implementation and experience in Spec#. Section 7 discusses possible extensions and future work, Section 8 contains related work, and Section 9 concludes.

2. Language

We use a small core of an imperative object-oriented language to formalize our ideas. The language consists of class declarations with fields and methods. We assume that the declaration information is provided by the overloaded class environment Ω as shown in Fig. 1. To avoid unnecessary complications, all methods are virtual instance methods and a method name m stands for a unique non-overriding method declaration. Thus, unrelated methods (not having a common base method) must have distinct names. $\Omega(m)$ therefore maps a method name to the unique declaring class and signature where m is introduced. A particular method definition in a class C is found via $\Omega(C.m)$. We assume similarly, that each field has a unique name and that fields cannot be shadowed or overwritten. Thus, $\Omega(f)$ maps a field name f to the declaring class and type of the field. Note that the declared field types need never be delayed and our language thus uses τ for field types, rather than possibly delayed types σ. We assume that the inheritance induced subtyping relation is provided by $\Omega \vdash C \leq D$ and that it is reflexive, transitive, and anti-symmetric.

Each method is universally quantified (generic) over a set of time variables $[\vec{t},\Gamma]$ that are constrained by an environment Γ. Additionally, method signatures consist of a return type σ_r, the method name and defining class $C.m$, and names and types of formal parameters $y_i : \sigma_i$. The receiver (self or this) has an explicit name and is always the first parameter y_0. The signature further consists of the effect E_r describing the set of fields initialized by the method. We use such effects to ensure the initialization of fields. This approach obviates the need to treat constructors specially.

Figure 2 describes the expression and type language. The expression language consists of standard parts such as local bindings, field reads and writes, and sequential composition. Method calls are standard except that they contain an explicit instantiation $[\vec{\chi}]$ of the quantified time variables. To reduce the number of cases in the calculus, all calls are virtual. Adding non-virtual calls does not pose any additional problems. The **ifnull** conditional allows testing reference values against **null**, which is the default value of all fields (we don't need an explicit **null** constant).

The **delay** t **in** e expression introduces a fresh time t in the scope of expression e. Allocation expression **let** $y =$ **alloc** $C[t]$ **in** e allocates a fresh object of class C that must be initialized by some future time t. In order to simplify the technical presentation, we explicitly name the allocated object with the binding y and enforce a stricter rule that requires the initialization of the object by the end of expres-

$$\frac{}{\Gamma \models_{\leq} \sigma \leq \sigma}\ [\text{ST-REFL}] \qquad \frac{\Omega \vdash C \leq D}{\Gamma \models_{\leq} C \leq D}\ [\text{ST-INHERIT}] \qquad \frac{\Gamma \models_{\leq} C \leq D}{\Gamma \models_{\leq} C! \leq D!}\ [\text{ST-NONNULL1}]$$

$$\frac{\Gamma \models_{\leq} C \leq D}{\Gamma \models_{\leq} C! \leq D}\ [\text{ST-NONNULL2}] \qquad \frac{\Gamma \models_{\leq} \tau_1 \leq \tau_2}{\Gamma \models_{\leq} \tau_1^{\Diamond\chi} \leq \tau_2^{\Diamond\chi}}\ [\text{ST-DELAY}] \qquad \frac{\Gamma \vdash_{\mathfrak{t}} t \leq \mathsf{Now} \quad \Gamma \models_{\leq} \tau_1 \leq \tau_2}{\Gamma \models_{\leq} \tau_1^{\Diamond t} \leq \tau_2^{\Diamond \mathsf{Now}}}\ [\text{ST-NOW}]$$

Figure 3. Subtyping rules

$$\frac{\Gamma \vdash_{\mathfrak{t}} \chi_1 \leq \chi_2 \quad \Gamma \vdash_{\mathfrak{t}} \chi_2 \leq \chi_3}{\Gamma \vdash_{\mathfrak{t}} \chi_1 \leq \chi_3}\ [\text{TM-TRANS}] \qquad \frac{\Gamma = \Gamma', t{\leq}\chi, \Gamma''}{\Gamma \vdash_{\mathfrak{t}} t \leq \chi}\ [\text{TM-NOLATER}] \qquad \frac{}{\Gamma \vdash_{\mathfrak{t}} \chi \leq \chi}\ [\text{TM-REFL}]$$

$$\frac{\Gamma = \Gamma', t{>}\mathsf{Now}, ..}{\Gamma \vdash_{\mathfrak{t}} t > \mathsf{Now}}\ [\text{TM-LATER}]$$

Figure 4. Timing rules

sion scope e. Due to syntactic nesting, this scope ends prior to the arrival of time t.

Finally, because we make use of existential types, we use an explicit **unpack** expression to bind the unpacked value and enforce scoping of the existentially bound time variable. We bind the existentially bound time variable to program time variable t.

Initialization effects E consist of a set of variable-field pairs. Ordinary types τ are either possibly null class references C, non-null references $C!$, or void (we do not add any other primitive types for simplicity). Delayed types σ consist of an ordinary type τ and a delay time χ. Time expressions χ are either the equivalence class Now of times that have arrived, or a time variable t. Finally, we also need existential quantification over constrained time variables in order to handle field reads. The form $\exists(t \leq \chi).\sigma$ binds the variable t.

Typing environments Γ contain both typing assumptions for expression variables y, as well as binding assumptions for time variables t. The latter come in three forms: 1) an unconstrained t (useful if a method is generic in t), 2) $t \leq \chi$, binding t with upper-bound χ, and 3) $t > \mathsf{Now}$, binding t as a future time.

Subtyping Subtyping (Fig. 3) in our system is restricted to subtyping induced by inheritance and subtyping between possibly null and non-null references. Timing relations do not induce subtyping due to the fact that fields of objects are mutable. The only rule that involves timing constraints is ST-NOW which axiomatizes the equivalence class Now. All times $t \leq \mathsf{Now}$ are considered equivalent to Now.

Timing The timing rules in Fig. 4 allow one to derive timing constraints from timing assumptions in the environment Γ.

2.1 Example Revisited

Returning to our example from Section 1, we can now type the initialization of the doubly linked list in our calculus using delayed references as follows:

```
1  List . sentinel  :  Node!
2
3  void  List . ctor [t, t>Now](this:List!^◇t) {this.sentinel}
4  {
5    let  tmp:Node!^◇t = alloc Node[t] in // allocate
6    tmp.ctor1[t]( this );                // call  constructor
7    this . sentinel  = tmp;
8  }
9
10 void  Node.ctor1[t, t>Now](this:Node!^◇t, parent:List!^◇t)
11   {this . parent ,  this . prev ,  this . next}
12 {
13   this . parent  = parent;
14   this . prev  = this;
15   this . next  = this;
16 }
```

Even though we will only present the type rules in the next section, we discuss here how the type system reasons about the above code. The List . ctor method is generic over a time t with constraint $t{>}$Now. The type of **this** is declared as List ! $^{\Diamond t}$, ie., it is also delayed with time t. The constraint $t{>}$Now is needed to allow the allocation of the sentinel Node on line 5, as allocations always require a future time. The allocation of the sentinel node yields a delayed reference of type Node!$^{\Diamond t}$ which is bound to tmp [1]. The choice of time t for the allocation of Node is crucial here. It specifies that both the List and its sentinel Node become valid at the same time. This allows them to safely refer to each other prior to being fully initialized. The delayed types of the List

[1] We show the type here, although the calculus does not need type annotations in bindings.

and the sentinel Node guarantee that no code can rely on the invariants of these objects prior to t. For example, were we to read the parent field prior to time t, we must read it through a delayed reference of type Node!$^{\diamond t}$. The type system will produce a type List$^{\diamond t'}$ for some delay t', stating that the parent reference itself could be null, and that the invariants of the parent may not hold either.

To guarantee that all the non-null declared fields of the sentinel node thus allocated are eventually initialized, the type system generates a proof obligation that all these fields are written to in the block following the allocation. This obligation is immediately discharged by the call to the ctor1 method, since this method is declared with initialization effect {parent,prev,next}, thus guaranteeing that these fields are written. The call itself is checked as follows: the method Node.ctor1 is also generic over a time t, which we instantiate here with the bound time t of List.ctor. The constraint $t>$Now is also satisfied. Then, the receiver tmp and the parameter **this** have exactly the types required by the Node.ctor1 signature (Node!$^{\diamond t}$ and List !$^{\diamond t}$).

Next, tmp is assigned to the sentinel field of **this**. Note that the sentinel node in tmp still has a delayed type Node!$^{\diamond t}$ at this point, but the sentinel field of List is simply Node!, i.e., not delayed. Yet, the assignment is allowed because the containing List object (**this**) is itself delayed with time t, which is obviously no sooner than the delay of the sentinel. The value written to the field must be non-null according to the field type, which is guaranteed by the type of tmp. Finally, at the end of method List.ctor, the return type and the initialization effects are checked. The method is declared to initialize sentinel, which is the case.

Now, consider some client code building a fresh empty list. In order to allocate, the code requires a future time t. A **delay** block is used to bind a fresh time t.

```
let list : List !^◇Now =
  delay t
  in
    let tmp : List !^◇t = alloc List[t] // allocate
    in
      tmp.ctor[t]();                    // call constructor
      tmp                               // result of delay
```

Within the delay scope, we can now allocate a List object and call its constructor to satisfy the proof obligation to initialize all non-null fields. The newly allocated list is the result of the delay block. Since at that point, the delay time t arrives, the type of the list can be changed to List !$^{\diamond \text{Now}}$ by renaming the bound time t to Now in the result of the delay block.

The next section presents all the type rules and describes them in more detail.

$$\frac{\begin{array}{l}\Omega(m) = \sigma_r\ D.m[\vec{t},\Gamma](y_0 : D!^{\diamond\chi_0}, y_1 : \sigma_1, .., y_n : \sigma_n)\ E_r \\ \Omega \vdash C \leq D \qquad \mathsf{bv}(\Gamma) = \vec{t} \\ \Gamma_0 = \Gamma, y_0 : C!^{\diamond\chi_0}, y_i : \sigma_i \qquad \bullet \vdash_{\mathsf{wf}} \Gamma_0 \\ \Gamma_0 \vdash_{\mathsf{e}} e : \sigma_r, E \\ E_r \subseteq \{(y,f) \mid (y,f) \in E \wedge \exists i.y = y_i\}\end{array}}{\Omega \vdash \sigma_r\ C.m[\vec{t},\Gamma](y_0 : C!^{\diamond\chi_0}, y_1 : \sigma_1, .., y_n : \sigma_n)\ E_r\ \{\ e\ \}}$$

Figure 7. Method typing

3. Typing

3.1 Method Typing

The method typing rule in Fig. 7 checks the well-formedness of a method. We do not explicate how the class environment is checked, but we assume that each method is checked via this rule. The rule states that in class environment Ω, method $C.m$ is well-typed, provided that method m's declared signature in class D is equivalent to the signature of $C.m$ (modulo the receiver's class), that C inherits from D (or is D), that the timing constraint environment Γ binds all quantified timing variables $\vec{t}$. More flexible overriding policies are of course possible (such as covariant return types, contravariant argument types), but such extension are orthogonal to the problem studied here.

The method body must be well typed in environment Γ_0, formed by augmenting the timing constraints Γ with bindings for the formal parameter types $y_i : \sigma_i$. Environment Γ_0 must be well-formed according to the rules of Fig. 5, which guarantees that each timing variable is bound exactly once and no unbound timing variables are used.

Finally, the last line of the antecedent guarantees that the declared initialization effect E_r is a subset of the initialization effect of the body e and that it only refers to the method parameters.

3.2 Expression Typing

Figure 8 shows the type rules for all expressions in our language. The judgments have the form $\Gamma \vdash_{\mathsf{e}} e : \sigma; E$, meaning that under assumptions Γ, expression e has type σ and effect E. Note that the type system does not take advantage of the effect E to reason about fields that might already have been initialized. The effect E is solely used to prove that all fields are eventually initialized after allocation and prior to the delay time. We'll explain each rule in turn.

Rule T-LOOKUP retrieves typing assumptions for bindings from the environment. It has no initialization effect. Rule T-LET is also standard except that the resulting effect E is filtered to not include any initializations on the bound variable y in the body e_2. Rule T-SUB is a standard subtyping rule.

Rule T-DELAY handles the introduction of a delay scope, binding future time t in expression e. Time t must not occur in the environment Γ. Body e is then typed with the addi-

$$\frac{}{\Gamma \vdash_{\mathsf{wf}} \tau}\ [\textsc{Wf-Type}] \qquad \frac{\Gamma \vdash_{\mathsf{wf}} \tau \qquad \Gamma \vdash_{\mathsf{wf}} \chi}{\Gamma \vdash_{\mathsf{wf}} \tau^{\diamond\chi}}\ [\textsc{Wf-DelayT}] \qquad \frac{\Gamma \vdash_{\mathsf{wf}} \chi \qquad \Gamma, t{\leq}\chi \vdash_{\mathsf{wf}} \sigma}{\Gamma \vdash_{\mathsf{wf}} \exists(t \leq \chi).\sigma}\ [\textsc{Wf-ExistT}]$$

$$\frac{}{\Gamma \vdash_{\mathsf{wf}} \mathsf{Now}}\ [\textsc{Wf-Now}] \qquad \frac{\Gamma = \Gamma', t, ..}{\Gamma \vdash_{\mathsf{wf}} t}\ [\textsc{Wf-Assume1}] \qquad \frac{\Gamma = \Gamma', t{\leq}\chi, ..}{\Gamma \vdash_{\mathsf{wf}} t}\ [\textsc{Wf-NoLater1}] \qquad \frac{\Gamma = \Gamma', t{>}\mathsf{Now}}{\Gamma \vdash_{\mathsf{wf}} t}\ [\textsc{Wf-Delay1}]$$

$$\frac{}{\Gamma \vdash_{\mathsf{wf}} \bullet}\ [\textsc{Wf-Empty}] \qquad \frac{\Gamma \vdash_{\mathsf{wf}} \Gamma' \quad t \notin \mathsf{bv}(\Gamma, \Gamma')}{\Gamma \vdash_{\mathsf{wf}} \Gamma', t}\ [\textsc{Wf-Assume2}] \qquad \frac{\Gamma \vdash_{\mathsf{wf}} \Gamma' \qquad \Gamma, \Gamma' \vdash_{\mathsf{wf}} \sigma}{\Gamma \vdash_{\mathsf{wf}} \Gamma', y{:}\,\sigma}\ [\textsc{Wf-Assume3}]$$

$$\frac{\Gamma \vdash_{\mathsf{wf}} \Gamma' \quad \Gamma, \Gamma' \vdash_{\mathsf{wf}} \chi \quad t \notin \mathsf{bv}(\Gamma, \Gamma')}{\Gamma \vdash_{\mathsf{wf}} \Gamma', t{\leq}\chi}\ [\textsc{Wf-NoLater2}] \qquad \frac{\Gamma \vdash_{\mathsf{wf}} \Gamma' \qquad t \notin \mathsf{bv}(\Gamma, \Gamma')}{\Gamma \vdash_{\mathsf{wf}} \Gamma', t{>}\mathsf{Now}}\ [\textsc{Wf-Delay2}]$$

Figure 5. Well-formedness

$$\mathsf{bv}(\bullet) = \langle\rangle \qquad \mathsf{bv}(\Gamma, t) = \mathsf{bv}(\Gamma), t \qquad \mathsf{bv}(\Gamma, t{\leq}\chi) = \mathsf{bv}(\Gamma), t$$
$$\mathsf{bv}(\Gamma, t{>}\mathsf{Now}) = \mathsf{bv}(\Gamma), t \qquad \mathsf{bv}(\Gamma, y{:}\,\sigma) = \mathsf{bv}(\Gamma), y$$

Figure 6. Bound variables

tional assumption that $t > \mathsf{Now}$. Finally, the resulting type is the type of the body σ, but with t replaced with Now, as the time t logically arrives at the end of the delay scope.

Rule T-ALLOC allocates a new object of class C with delay t and binds it to y in the body e. Time t must strictly be in the future, as the newly allocated object does not satisfy the invariants of non-null fields. Type-checking block e yields initialization effect E, which must cover all the non-null fields of C and its base classes ($\mathsf{nnfields}(C)$). Thus, at the end of the allocation block e, the newly allocated object satisfies its field invariants. This is sufficient to prove that at the end of the delay scope for t, all fields of all objects with $o.\mathsf{ValidTime} = t$ are initialized.

The conditional rule T-IFNULL is a standard conditional rule, except that the condition variable y is rebound in the else branch to a non-null type $C!$, and the resulting initialization effect is the intersection of the initialization effects of both branches. The type of the condition may be delayed for any time t, as the conditional only examines the pointer value, not the object's fields.

Writing to a field is handled by rule T-WRITE. Given that y has type $C!^{\diamond t_1}$, and field f is declared in C or one of its base classes, the assignment is well-typed, provided that additionally, the assigned type is compatible with the field type and the delay t_2 of the stored value is no later than the delay of the container t_1. This is the crucial rule maintaining the necessary invariant that all objects reachable from a reference with delay t have delays no later than t. The rule also establishes an initialization effect for field f of binding y.

Handling field reads T-READ is probably the most complicated aspect of the type system. Given the invariant just mentioned, we do not actually know the delay t_2 of a reference stored in a field f. Furthermore, the field type τ might have to be weakened to include **null**, unless we can prove that the container is not delayed. We thus do a case-split. The result type σ of the read is $\tau^{\diamond\mathsf{Now}}$, if we can prove that the container is not delayed. In this case, the field type is the declared field type and the read value is not delayed. Otherwise, we produce an existentially quantified type, where we abstract over delay t_2. We only know that delay t_2 is no later than the delay t_1 of the containing object. We further weaken the field type τ to $\tau?$, which has the effect of removing any non-nullness.

Rule T-UNPACK is a standard existential elimination rule. It types the body e_2 of the scope under the assumption that the existentially bound time t with constraint $t \leq \chi$, while guaranteeing that t is fresh and does not escape the scope e_2.

Finally, rule T-CALL handles method calls. It types the receiver and all arguments and forms the substitution φ that maps formal time parameters $\vec{t}$ to actual times $\vec{\chi}$. The judgment $\Gamma \vdash_{\varphi} \Gamma'$ ensures that the assumptions Γ imply the assumptions Γ' under the substitution φ. The rules for implication are shown in Fig. 9. If all actual argument types match the formal argument types under substitution φ, the call is well-typed and results in effects E_r, where we substitute the actual bindings y_i for the formals y_i'.

4. Correctness

This section argues the correctness of our approach. Consider the delay scopes active at runtime. These scopes parti-

$\boxed{\Gamma \vdash_{\mathsf{e}} e : \sigma; E}$

$$\frac{\Gamma = \Gamma', y{:}\,\sigma, ..}{\Gamma \vdash_{\mathsf{e}} y : \sigma; \emptyset}\ [\textsc{T-Lookup}] \qquad \frac{\begin{array}{l}\Gamma \vdash_{\mathsf{e}} y : C^{\diamond t}; \emptyset \\ \Gamma \vdash_{\mathsf{e}} e_1 : \sigma; E_1 \\ \Gamma, y{:}\,C!^{\diamond t} \vdash_{\mathsf{e}} e_2 : \sigma; E_2 \\ E = E_1 \cap E_2\end{array}}{\Gamma \vdash_{\mathsf{e}} \textbf{ifnull}\, y\, \textbf{then}\, e_1\, \textbf{else}\, e_2 : \sigma; E}\ [\textsc{T-IfNull}]$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{e}} e : \sigma_1; E \\ \Gamma \vdash_{\leq} \sigma_1 \leq \sigma_2\end{array}}{\Gamma \vdash_{\mathsf{e}} e : \sigma_2; E}\ [\textsc{T-Sub}] \qquad \frac{\begin{array}{l}t \notin \mathsf{bv}(\Gamma) \\ \Gamma, t{>}\mathsf{Now} \vdash_{\mathsf{e}} e : \sigma; E\end{array}}{\Gamma \vdash_{\mathsf{e}} \textbf{delay}\, t\, \textbf{in}\, e : \sigma[\mathsf{Now}/t]; E}\ [\textsc{T-Delay}]$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{e}} e_1 : \sigma_1; E_1 \\ \Gamma \vdash_{\mathsf{e}} e_2 : \sigma_2; E_2 \\ E = E_1 \cup E_2\end{array}}{\Gamma \vdash_{\mathsf{e}} e_1; e_2 : \sigma_2; E}\ [\textsc{T-Seq}] \qquad \frac{\begin{array}{l}\Gamma \vdash_{\mathsf{t}} t > \mathsf{Now} \\ \Gamma, y{:}\,C!^{\diamond t} \vdash_{\mathsf{e}} e : \sigma; E_1 \\ \forall f \in \mathsf{nnfields}(C).(y, f) \in E_1 \\ E = E_1 \setminus \{(y, \_)\}\end{array}}{\Gamma \vdash_{\mathsf{e}} \textbf{let}\, y = \textbf{alloc}\, C[t]\, \textbf{in}\, e : \sigma; E}\ [\textsc{T-Alloc}]$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{e}} y : C!^{\diamond t_1}; \emptyset \\ \Gamma \vdash_{\mathsf{e}} e : \tau^{\diamond t_2}; E_1 \\ \Omega(f) = C.f : \tau \\ \Gamma \vdash_{\mathsf{t}} t_2 \leq t_1 \\ E = E_1 \cup \{(y, f)\}\end{array}}{\Gamma \vdash_{\mathsf{e}} y.f := e : \mathsf{void}; E}\ [\textsc{T-Write}] \qquad \frac{\begin{array}{l}\Gamma \vdash_{\mathsf{e}} e : C!^{\diamond t_1}; E \quad \Omega(f) = C.f : \tau \\ \sigma = \begin{cases} \tau^{\diamond \mathsf{Now}} & \text{if } \Gamma \vdash_{\mathsf{t}} t_1 \leq \mathsf{Now} \\ \exists (t_2 \leq t_1).\tau?^{\diamond t_2} & \text{otherwise} \end{cases}\end{array}}{\Gamma \vdash_{\mathsf{e}} e.f : \sigma; E}\ [\textsc{T-Read}]$$

where $(C!)? = C$ and $C? = C$.

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{e}} e_1 : \exists (t \leq \chi).\sigma_1; E_1 \\ t \notin \mathsf{bv}(\Gamma) \qquad t \notin \mathsf{fv}(\sigma_2) \\ \Gamma, t{\leq}\chi, y{:}\,\sigma_1 \vdash_{\mathsf{e}} e_2 : \sigma_2; E_2 \\ E = E_1 \cup (E_2 \setminus \{(y, \_)\})\end{array}}{\Gamma \vdash_{\mathsf{e}} \textbf{unpack}\, t, y = e_1\, \textbf{in}\, e_2 : \sigma_2; E}\ [\textsc{T-Unpack}] \qquad \frac{\begin{array}{l}\Gamma \vdash_{\mathsf{e}} e_1 : \sigma_1; E_1 \\ \Gamma, y : \sigma_1 \vdash_{\mathsf{e}} e_2 : \sigma_2; E_2 \\ E = E_1 \cup (E_2 \setminus \{(y, \_)\})\end{array}}{\Gamma \vdash_{\mathsf{e}} \textbf{let}\, y = e_1\, \textbf{in}\, e_2 : \sigma_2; E}\ [\textsc{T-Let}]$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{e}} y_i : \sigma_i; \emptyset \quad i = 0..n \\ \Omega(m) = \sigma_r\ D.m[\vec{t}, \Gamma'](y'_0 : \chi'_0, y'_1 : \sigma'_1, .., y'_n : \sigma'_n)\ E \\ \sigma'_0 = D!^{\diamond \chi'_0} \qquad \varphi = [\vec{\chi}/\vec{t}] \qquad \Gamma \vdash_{\varphi} \Gamma' \\ \Gamma \vdash_{\leq} \sigma_i \leq \varphi(\sigma'_i) \quad i = 0..n\end{array}}{\Gamma \vdash_{\mathsf{e}} y_0.m[\vec{\chi}](y_1, .., y_n) : \varphi(\sigma_r); E[y_i/y'_i]}\ [\textsc{T-Call}]$$

Figure 8. Typing Rules

$$\frac{}{\Gamma \vdash_{\varphi} \bullet}\ [\textsc{Imp-Empty}] \qquad \frac{\Gamma \vdash_{\mathsf{wf}} \varphi(t) \qquad \Gamma \vdash_{\varphi} \Gamma'}{\Gamma \vdash_{\varphi} \Gamma', t}\ [\textsc{Imp-Bind}]$$

$$\frac{\Gamma \vdash_{\mathsf{t}} \varphi(t) \leq \varphi(\chi) \qquad \Gamma \vdash_{\varphi} \Gamma'}{\Gamma \vdash_{\varphi} \Gamma', t{\leq}\chi}\ [\textsc{Imp-NoLater}] \qquad \frac{\Gamma \vdash_{\mathsf{t}} \varphi(t) > \mathsf{Now} \qquad \Gamma \vdash_{\varphi} \Gamma'}{\Gamma \vdash_{\varphi} \Gamma', t{>}\mathsf{Now}}\ [\textsc{Imp-Delay}]$$

Figure 9. Constraint Implication

tion the heap objects into disjoint time regions, one per delay scope, plus the non-delayed region Now, as depicted in Figure 10. Each object o belongs to the region designated by $o.\mathsf{ValidTime}$. When an object is allocated with, say, delay t_n, it enters the region associated with that delay time. When the delay scope t_n ends, the region is joined to the non-delayed region Now and the delay time t_n becomes equivalent with Now.

Objects can refer to other objects within the same time region, or can refer to objects in regions with an earlier delay time (upward in the figure). Delay times are ordered according to the delay scopes entered at runtime. The delay scopes form a stack, where the last scope entered will be exited prior

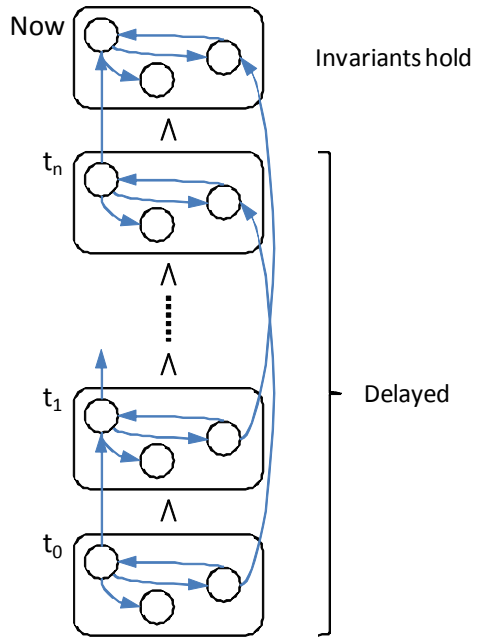

Figure 10. Time regions and their relation

to all scopes below it. In the picture, delay scope t_0 is the outermost delay scope, and t_n is the currently innermost scope. As execution progresses, either new nested scopes are added between t_n and Now, or t_n is exited, merging t_n's region to the Now region of non-delayed objects.

The following invariant captures the reference pattern described in Figure 10.

INVARIANT 4.1. *Objects only point to other objects with same or smaller delays*

$$\forall o.o \rightsquigarrow p \implies p.\mathsf{ValidTime} \leq o.\mathsf{ValidTime}$$

where $o \rightsquigarrow p$ means that object p is reachable from o through 0 or more field references.

The next invariant captures the fact that delayed objects need not satisfy their field invariants yet.

INVARIANT 4.2. *Non-delayed objects satisfy their field invariants.*

$$\forall o.(o.\mathsf{ValidTime} \in \mathsf{Now}) \implies \forall(f \in \mathsf{nnfields}(o)).o.f \neq \textbf{null}$$

We now argue how these invariants are maintained. First, consider invariant 4.1. Only three operations in the language influence this formula: allocation, delay, and field assignment. Allocating a new object adds a new element to the quantification. The property holds trivially for the new object, as it does not reference, nor is it referenced by any other object. Exiting a delay scope joins the time of the region on the top of the stack (t_n) to Now. This can be viewed as changing ValidTime of each object in t_n to Now. Consider each reference from an object in t_n. The target of such a reference can only be in Now or in t_n itself. After the change, these refer from objects in Now to objects in Now, which is admissible. Consider a reference targeting an object in t_n. After the change, this reference targets an object in region Now. Such references are allowed from all regions, thus all such references are still valid after the change.

Finally, updating a field requires proving that the delay of the stored value is less-than or equal to the delay time of the container. Type rule T-WRITE enforces this.

Of course, we must also show that the delayed types conservatively approximate the actual ValidTime of the typed object. At allocation, this is obviously the case. Inspecting the type rules shows that the delay in the type of a value can only be changed by storing the value into a field and reading it back. Storing a value with delay t_1 into a field of an object with delay t_2 amounts to existential abstraction (forgetting the exact time) of t_1. This is visible in field reads, where the delay time of the read object is simply some time $t \leq t_2$.

Now consider invariant 4.2. This formula is affected by the same three operations: allocation, delay, and field update. Allocation adds a new element to the domain of the quantification. Because allocation requires a delay time $t >$ Now, strictly later than Now, we guarantee that the implication is trivially satisfied. When ending a delay scope, all objects in t_n now need to satisfy the right-hand side of the implication. Fortunately, the allocation rule checks that all fields are written within the scope of the allocation (and thus within the scope binding the delay time). This, together with the check in type rule T-WRITE guaranteeing that if the field type is non-null, the written value is also non-null, guarantees that all non-null declared fields are indeed non-null by the end of the delay scope. The field update type rule does not permit writing null into a field that is declared non-null. Thus, once a non-null typed field is non-null, it can never revert back to null.

To guarantee that $t >$ Now implies $t \notin$ Now, we must be careful that the constraint $t >$ Now cannot survive the delay scope binding t. Rule T-DELAY enforces that t is not free in the environment or the result and our existential types cannot capture $t >$ Now constraints.

Finally, it remains to verify that non-null types correctly approximate the non-nullness of the typed values. Non-null types are introduced only by the allocation rule (we assume allocation always succeeds). Inspecting subtyping and the type rules shows that no other means of obtaining a non-null type exist except reading a field with a non-null type declaration. Rule T-READ yields only a non-null type given that the container is not-delayed. Otherwise, the result is of possibly null type, mirroring invariant 4.2.

4.1 Multithreading discussion

It is noteworthy that the delayed approach extends naturally to a multi-threaded setting. One simply has to ensure that delayed objects are never shared. This is natural and simple to enforce: 1) delayed objects can never be stored in global locations accessible by multiple threads. 2) delayed objects

cannot be stored in fields of already shared objects (already enforced by field update rule), as the shared object cannot be delayed, and 3) starting a new thread requires all parameters to the thread to be non-delayed objects.

5. Spec# Implementation

We have implemented the idea of delayed types in Spec#, an experimental extension of C# with non-null types, pre-conditions, post-conditions, and invariants [1]. The next subsections first describe the existing mechanism in Spec# to deal with non-null types on fields, then sketch how we add the idea of delayed types. We show how Spec#-level constructs map to the formalization of the previous sections and how the proof obligations are discharged at the high-level. We also discuss how to handle features of the language not described in the calculus.

5.1 Prior approach to non-null typed fields

Prior to the introduction of delayed types, Spec# used the following approach to guarantee proper initialization and avoiding exposure of non-initialized objects during construction: constructors of objects with non-null typed fields must initialize all such fields *prior* to the base constructor call, and the only allowed operation on the object under construction prior to the base constructor call is to write its fields. This simple approach guarantees that in each constructor, after the base constructor call, all non-null typed fields of the object are non-null, thereby avoiding the need for raw types.

The drawbacks of this approach are as follows: 1) new syntax is needed in the constructor to enable initialization of fields as a function of constructor arguments prior to the base constructor call. In C++ e.g., special syntax exists to initialize fields. In Spec#, we opted to make the base constructor call explicit in the body of constructors as **base** (...) . The language requires that there is a base constructor call on each path through the constructor. 2) as a result, every constructor of a class with non-null typed fields must have an explicit base constructor calls in its body. This can be a nuisance when porting existing code from C# and changing some field types to non-null, as it most often requires putting a base constructor call at the end of the method. Sometimes, initialization of fields in constructors is interspersed with calls on the object under construction, and in order to place the base constructor call, further reordering of the code is needed. 3) circular initialization patterns as in our example in the introduction, cannot be dealt with at all.

With the introduction of delayed types, most constructors ported from C# can remain unchanged, even after changing some field types to non-null types, as the initialization of these fields may now occur after the base constructor call. Furthermore, if there are methods called on the object under construction, it is often possible to mark those methods as taking a delayed receiver, without the need to rearrange the constructor code.

5.2 Delayed Types

In Spec#, we use an attribute [Delayed] instead of the low-level types of the formalization. The attribute can only appear in method signatures on parameters, method results, and the method itself. In the last case, it applies to the method's receiver (**this**). We interpret these attributes as follows: We implicitly quantify each method over a single time variable t. All types in the method signature attributed with [Delayed] are delayed by t. All non-attributed types are not delayed (Now). We found that being able to mention a single delay time seems sufficient in most examples. Using our attribute syntax, we avoid ever naming a time explicitly.

5.3 Constructors

Constructors are special in that the implicit **this** parameter is by default delayed with the implicit time parameter t. Furthermore, we assume that constructors have an implicit effect clause that specifies the initialization of all non-null fields (including those of base classes). This, together with base calls and the type rules in Sect. 3 guarantees that all non-null fields are initialized by the time the constructor returns.

Finally, in constructors, the implicitly quantified time variable t has bound $t > \mathsf{Now}$, whereas in ordinary methods it is unconstrained.

For backward compatibility with Spec#'s prior initialization scheme, we support the annotation [NotDelayed] on constructors. This annotation specifies that the receiver is not to be treated as delayed, but that all non-null fields have to be initialized prior to the base constructor call as discussed above. Thus, programmers have a choice whether to treat constructors as delayed or not delayed, resulting in one of two initialization schemes. Non-delayed constructors are most useful when the object under construction is used for significant computation after initialization prior to returning from the constructor.

5.4 Allocation

Each allocation in the high-level language **new** C(...) is modeled in the calculus as

delay t **in** **let** tmp = **alloc** C[t] **in** tmp.ctor[t](...); tmp

Together with the default initialization effect of constructors, this translation guarantees that the type rule for alloc is always satisfied.

To allocate at an existing delay, the high-level construct **new** C[Delayed](...) is used. It translates to

let tmp = **alloc** C[t] **in** tmp.ctor[t](...); tmp

where t is the implicit time parameter of the current method. As only constructors make the assumption that $t > \mathsf{Now}$, this construct is only supported in constructors.

We can reason how the high-level language and its rules guarantee the necessary invariants at the low-level.

- Each constructor is checked to initialize all its non-null fields. Together with the forced call to the base constructor, this is sufficient to establish the implicitly specified effect of constructors (which is that all non-null typed fields of the **this** object are non-null).
- The rule for field assignment guarantees the reachability invariant (4.1).
- Finally, the proof obligation at the end of an allocation block is satisfied by construction of the translation sketched above, as each **alloc** block contains a constructor call on the newly allocated object.

5.5 Existential Types

We do not want to deal with existentials at the Spec# language level. Observe that existentials only arise when reading from a delayed reference. Existentially quantified types simply stand for references with unknown delay. We can wrap each access to such references within implicit unpack expressions and weaken the type by forgetting the constraint on the existentially bound variable. This weakening results in some loss of expressiveness, but in practice, it has not been a problem. It precludes for example reading a reference from a delayed object, and then storing it back into the same object.

5.6 Exceptions

Exceptions complicate the checking of the initialization effects in allocation blocks. When an exception escapes an allocation block, the fields of the newly allocated object with delay time t, might not yet be properly initialized. Yet, the allocated object can escape the block by virtue of having been assigned to another delayed object allocated earlier. If the exception is handled within the delay scope for t, then the object might escape the delay without proper initialization. We can prevent this scenario by disallowing handling of exceptions in constructors. This way, it is guaranteed that an exception raised in an allocation scope unwinds all current delay scopes.

5.7 Generics

Generics in C# or Java provide universal quantification over types. This quantification has to be done independently of the quantification over delay time. Quantification over a type cannot include its delay. Expressed in our formalism, type quantification would range over τ, not over σ. Thus, a method might have a formal parameter with type T, where T is a type parameter, yet the delay of the parameter type is specified independently.

5.8 Delegates

Delegates in C# are method closures capturing an object and a method to call on that object. The constructor of delegates takes the captured object as a parameter (along with the method pointer). We found that often delegates were constructed by capturing a delayed object. To support this scenario, the delegate constructor signature could be given delayed types for the parameter and the receiver. However, because we only support one quantified time variable, such a signature makes the constructor unusable to capture a non-delayed object. Using a separate delay time t_1 for **this** and another $t_2 \leq t_1$ for the captured object makes such constructors applicable in all contexts.

This problem arises mainly because the delegate constructors are generated automatically. In programmer written constructors, one can work around this limitation by providing two separate constructors.

5.9 Initialization Helper Methods

Our current implementation requires that all non-null typed fields are initialized in constructors themselves. The calculus is more expressive in that it allows initialization to be done in helper methods called from constructors, if the helpers have the appropriate initialization effect specifications. We plan to add such specifications to Spec# in the future.

6. Experience

After implementing support for delayed types in Spec#, we adapted two substantial code bases to the new initialization scheme: Boogie, the verifier of Spec# and a large fraction of the Singularity research operating system. Boogie is about 50000 lines and 418 classes, the relevant Singularity code base is about 150,000 lines with 2003 classes, both including comments and white-space. Table 1 contains usage counts related to non-null and delayed types that illuminate the applicability of the proposed approach.

More than half the classes in Boogie and about a third of the classes in Singularity use some non-null field invariants. For these classes, over 91% of constructors in Boogie and over 95% in Singularity can be typed as delayed, meaning they do not need any special initialization syntax for the fields, or explicit placement of a **base** constructor call.

Maybe surprisingly, for classes without non-null typed fields, there are also some uses of non-delayed constructors. These arise due to inheritance and calls to non-delayed base constructors.

As we hoped, the number explicit uses of [Delayed] annotations is small in both code bases, even though the number of implicit delayed receivers for delayed constructors is 445 in Boogie and 1617 in Singularity. Our defaults for constructors thus work out well. We break down the explicit use of delayed into three categories: 1) We found 24 methods that are called from constructors, passing the object under construction as a parameter. These are often helper methods such as assertion methods, or property getters. 2) We found 16 uses of delayed parameters due to actual circular initializations. In Boogie there is one such use, in Singularity 15

	Boogie	Singularity
Classes	418	2003
with non-null typed fields	244	608
delayed ctors	266	671
non-delayed ctors	24	29
without non-null typed fields	174	1395
delayed ctors	179	946
non-delayed ctors	8	29
Explicit use of Delayed	2	43
call from ctor	1	23
circular init	1	15
delay alloc		5
Explicit use of **base** in ctor	37	N/A
computation after init	10	N/A
base ctor non-delayed	19	N/A
shared data between this and base	4	N/A
other	4	N/A
with prior initialization scheme	169	N/A

Table 1. Usage of delayed types

such uses. The simplest use is that of a parent object constructing a helper object that contains a back pointer to the parent, similar to our example. The most complicated form are several doubly linked list implementations. 3) Finally, we found 5 explicit uses of delayed allocations (all other delayed allocations are inferred).

For Boogie, we further studied the impact of delayed constructors on the use of explicit base constructor calls **base** (...) in constructor bodies, which was required in the prior scheme to allow initialization of the non-null typed fields prior to the base constructor call. Of the 169 uses of such explicit **base** (...) calls in constructor bodies prior to delayed type use, all but 37 where eliminated. We classified the remaining 37 uses into 4 categories: 1) we found 10 uses of non-trivial computations performed in the constructor or in methods called from the constructor after initialization is complete. 2) in 19 cases, the explicit base call was used because the base constructor was not delayed and the deriving class had non-null field initializations. 3) in 4 cases, we found that programmers used the more expressive power of explicit base calls in the body to compute data prior to the base call that was used both as an argument to the base call, as well as in the constructor itself. 4) finally, four uses were due to special constant fields used in the Boogie methodology itself.

Overall, the conversion was painless, due to good defaults.

7. Extensions

7.1 Arrays

We are currently working on extending the delay approach to handle arrays of non-null element types. For arrays, the elements of the array have to be treated like fields of an object. The main difference between arrays and object construction is that there is no well-delimited construction method for arrays, and the number of elements is not constant. One simple approach we are considering is to require an explicit runtime check via a pre-defined method that guarantees the initialization of all elements to non-null values. The approach described in this paper is then adapted to merely check that this runtime check is present on each path from an array allocation to the end of its delay scope.

7.2 Generalization

So far we have shown how to reason about invariants involving non-null fields. Our approach can be generalized to reason about more complicated invariants. In particular, the type system can directly support heap-monotonic predicates [5], i.e., properties, that once established, are never violated in future heaps.

7.3 Other Applications

The techniques sketched here can also be applied to functional programming languages, such as OCaml in order to generalize the allowable expressions on the right-hand-side of recursive bindings. The language would have to provide a form of constructor functions for tuples and datatype values that assign to the fields of the value under construction. Besides the automatically generated ones such as initACons, such constructor functions could then also be programmer

written. Reading from delayed values would have to be restricted to matching of the datatype tag only—no access to the arguments of datatypes or tuples would be allowed as there is no suitable null value.

7.4 Phased Initialization

One recurring problem in dealing with object invariants is that often, objects and entire object graphs are initialized in several phases. Only after the last phase are all invariants established. Consider for example a parser that builds ASTs bottom-up. It is possible that the programmer would like each node to point to its parent node (except for the root obviously). The parent pointers have to be established as a second phase.

Our technique so far cannot yet deal with this problem. However, we think it is a step in the right direction. Observe that objects allocated with delay t_n can only escape the delay scope for t_n by being reachable from the result of the delay scope expression. Thus, we have a guarantee that these objects are not reachable from anywhere else. One can exploit this observation by associating a new obligation with the result of a delay scope and instead of renaming t_n to Now, renaming it to t_{n-1}. Such a renaming also preserves our reachability invariants. The additional complication is that the new obligation must cover all objects with delay t_n, not just the top-level one. Given this idea, it is possible to associate new obligations to an object (graph) whenever it escapes from a delay scope. We plan to explore this idea further.

8. Related Work

The problem of properly initializing data structures has been studied in the context of compiling typed high-level languages to typed assembly language. At some point during the translation, memory allocation and initialization becomes explicit and the initialization steps must be captured by the type system. For this purpose, Morrisett et. al. developed an *alias type* system where the types of memory locations can change as these locations are updated [10, 13].

Alias types can be used to solve the initialization problem as long as detailed post conditions are provided. In particular, for our doubly linked-list example, the alias type solution would include specifying the exact pointer relations established by the Node constructor. Furthermore, the alias type approach would force the programmer to be explicit about aliasing (resp. non-aliasing). The same comments would apply if we were to use Hoare style reasoning with separation logic.

Our approach using delayed types differs from the above in that we do not have to reason about aliasing and we only require very abstract specifications about how fields are initialized. This makes the approach amenable for inclusion in a mainstream language with good defaults as described in the previous section.

Our work does not need to address the problem of proving the well-foundedness of recursive definitions [9, 3], as our circular structures always involve heap objects. Similarly, we do not try to address the issue of how to define mutually referential structures in the presence of abstract creation functions [11].

There are similarities of our work with the region type system of Tofte and Talpin [12]. A delay time t acts similarly to a region. A distinguishing aspect, however, is that at the end of a delay scope, a time region is not deleted, but joined with the Now region. The reachability invariant of references in our delay scopes seems superficially similar to the outlives relation of regions in the safe-C dialect Cyclone [6]. Interestingly, their pointer relation is inverse to ours, as references can point from newly allocated regions to objects in older regions, but not vice-versa.

Non-null types are now being considered for a number of languages besides Spec#, e.g., for JML-based Java [2]. These languages have to deal with the problem of observing non-null field invariants prior to their establishment. Raw types from our prior work [4] provide one solution. The present work improves substantially over that by providing the same guarantees, while supporting vastly more expressive initialization patterns, including circular structures. We thus advocate the use of delayed types in these language efforts over the inclusion of raw types.

9. Conclusions

We described a new approach to the initialization problem in OO-languages equipped with non-null types. The main idea is to include in the type system a notion of future time at which objects under construction become valid. References to objects under construction are typed with delayed types $\tau^{\diamond t}$, capturing that the object's field invariants may not hold prior to time t. This formalization allows a flexible field update rule that can store pointers with delayed types into fields of other delayed objects, as long as the containing object's delay is longer than the delay of the field value. Furthermore, there is no need to cater for this situation in the declared type of fields. The resulting type system is powerful enough to successfully type check cyclic initializations. We have implemented these ideas in the Spec# compiler. Our experience shows that with delayed types, a language supporting non-null typed fields can mostly avoid requiring field initializations prior to the base-constructor call. This is important for porting existing code in C# to Spec#, and similarly for porting Java code into JML [7]. In the latter case, the fact that constructor semantics do not need to change is of special importance, since the JML specifications live in source code comments and do not influence the code generation, which is performed by a normal Java compiler.

References

[1] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# Programming System: an Overview. In *Proceedings of the Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart devices*, volume 3362 of *LNCS*, 2004.

[2] P. Chalin and P. 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] Derek Dreyer, Robert Harper, and Karl Crary. A Type System for Well-Founded Recursion. In *Proceedings of the 31st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2004.

[4] Manuel Fähndrich and K. Rustan M. Leino. Declaring and Checking Non-Null Types in an Object-Oriented Language. In *Proceedings of the 2003 ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2003*, pages 302–312. ACM, November 2003.

[5] 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.

[6] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based Memory Management in Cyclone. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 282–293. ACM, May 2002.

[7] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.

[8] The OCaml Programming Language. http://caml.inria.fr.

[9] Claudio Russo. Recursive Structures for Standard ML. In *Proceedings of the 2001 International Conference on Functional Programming*, pages 50–61, 2001.

[10] Frederick Smith, David Walker, and Greg Morrisett. Alias types. In *Proceedings of the 14th European Symposium on Programming*, volume 1782 of *LNCS*, pages 366–381. Springer, March 2000.

[11] Don Syme. An Alternative Approach to Initializing Mutually Referential Objects. Technical Report MSR-TR-2005-31, Microsoft Research, March 2005.

[12] Mads Tofte and Jean-Pierre Talpin. Implementation of the Typed Call-by-Value lambda-Calculus using a Stack of Regions. In *Proceedings of the 1994 Symposium on Principles of Programming Languages*, pages 188–201, 1994.

[13] David Walker and Greg Morrisett. Alias types for recursive data structures. In *Proceedings of the 4th Workshop on Types in Compilation*, September 2000.