Heap Monotonic Typestates (Extended Abstract)

Manuel Fähndrich and K. Rustan M. Leino

Microsoft Research
{maf,rustan}@microsoft.com

Abstract. The paper defines the class of *heap monotonic typestates*. The monotonicity of such typestates enables sound checking algorithms without the need for non-aliasing regimes of pointers. The basic idea is that data structures evolve over time in a manner that only makes their representation invariants grow stronger, never weaker. This assumption guarantees that existing object references with particular typestates remain valid in all program futures, while still allowing objects to attain new stronger typestates. The system is powerful enough to establish properties of circular data structures.

1 Introduction

Types are the main mechanism by which programmers specify properties about data structures that are mechanically checked by today's compilers. Types, however, are a very limited specification tool, in particular in imperative programming languages, where objects evolve over time. As objects evolve, they acquire more properties, and stronger invariants get established. But such new properties cannot be captured in the form of types, since types in mainstream languages capture only properties that hold uniformly from the birth to the demise of an object.

This paper presents a statically checkable *typestate* system. Typestates [9] specify extra properties of objects beyond the usual programming language types. As the name implies, typestates capture aspects of the state of an object. When an object evolves, its typestate may evolve as well. Typestates can be used to restrict valid parameters, return values, or field values, and in doing so provide extra guarantees on internal object invariants.

The main contribution of this paper is that it identifies a class of *heap monotonic typestates*, along with a statically checkable condition on field updates. Under that condition, it can be proven that all object states evolve monotonically: statically observable object invariants only become stronger as objects evolve. At first, the idea may seem restrictive, but it results in a surprisingly liberal programming methodology: our static typestate discipline captures gradual initialization of entire object graphs, and can even prove properties of cyclic structures. A surprising result is that only typestate annotations are needed: there is no need for non-aliasing annotations or assumptions, nor is there a need

to declare read or write effects of methods. These properties put our approach at a minimal distance from ordinary type checking and distinguish it from previous related work on proving stronger program invariants [4, 3, 8, 1]. We believe this system is practical, because it puts no restrictions on the shape of object graphs. As a result, we expect our approach easily to combine with existing approaches for structuring the heap or managing resources, such as ownership types [2] or alias types [10]. Moreover, our typestate system is formulated in such a way that it can take advantage of non-aliasing information, if present.

The rest of the paper is organized as follows: Section 2 introduces heap monotonic typestates by means of an example. Section 3 formalizes typestates in the presence of inheritance and gives sufficient field update conditions to maintain monotonicity. Section 4 discusses further ramifications of monotonic typestate and future extensions. The remaining sections discuss related work and conclude.

2 Motivating example

To illustrate evolving objects, the example in Fig. 1 contains code fragments of a typical compiler front-end. The main data structure is an abstract syntax tree (AST) consisting of AstNode objects. The parsing, name resolution, type checking, and back-end phases are represented. Parsing produces an abstact syntax tree. This tree is then modified by first doing name resolution (ast.ResolveNames(...)), followed by doing type checking (ast.TypeCheck(...)). After type checking, the AST is passed to the back-end (ast.Emit(...)) .

An AST cannot be passed to the back-end without first performing type checking. Similarly, type checking cannot be performed without name resolution. The state of the AST after parsing therefore differs from its states after name resolution and after type checking. Mainstream programming languages do not allow programmers to express such state properties, let alone statically check them. Typestate annotations and typestate checking fill this gap.

We distinguish three states of the AST: "Naked" after parsing, "Bound" after name resolution, and "Typed" after type checking. Figure 1 lists typestate annotated signatures of the methods representing the various front-end phases. Annotation [return:Post("Naked")] states that method Parse returns an AST satisfying state "Naked". The methods performing name resolution, type checking, and code emission are instance methods of AstNode. They have Pre and/or Post annotations to express the typestate expected on entry, and the typestate guaranteed on exit. For instance, the annotation on method ResolveNames specifies that on entry, the receiver (this object) must satisfy state "Naked"; whereas on exit, it will satisfy state "Bound". The annotation on Emit simply requires the receiver to satisfy state "Typed".

So far, we have only seen typestates in their abstract form, that is, as adjectives modifying a type. In that form, typestate annotations may constrain when types are compatible. The other major purpose of typestates is to capture extra data invariants. Figure 2 shows class fragments of subclasses of AstNode with annotations expressing such invariants. In our example, the typestate "Typed"

```
void Main(...) {
  AstNode ast= Parse(filename);
  ast.ResolveNames(emptyEnvironment);
  ast.TypeCheck(emptyTypeEnvironment);
  ast.Emit (...);
}

[return:Post("Naked")] AstNode Parse(string file );

abstract class AstNode {

  [Pre("Naked"),Post("Bound")]
  abstract void ResolveNames(Env env);

  [Pre("Bound"),Post("Typed")]
  abstract void TypeCheck(TypeEnv typeEnv);

  [Pre("Typed")]
  abstract void Emit (...);

  ...
}
```

Fig. 1. Front-end of a compiler

captures the fact that the type field of Expression objects has been initialized to a non-null pointer to a Type object. This invariant is expressed by the annotation [NotNull(WhenEnclosingState="Typed")] on field type of class Expression. Similarly, the typestate "Bound" captures the fact that the binding field of Identifier objects in the AST has been initialized to a non-null pointer to the binding node.

These two annotations describe the relation between the typestate of an object and the *atomic properties* of its fields (in this case non-nullity).

Furthermore, we may relate the typestate of an object to the typestates of objects pointed to in its fields. For example, the UnaryExpr class needs to specify the state of the operand sub-expression in the AST. This state is dependent on the state of the unary expression node itself. In our example, the relation is simple: if the unary expression object satisfies state "Naked" (resp. "Bound", "Typed"), then so does the sub-expression. The annotation [InState("Naked", WhenEnclosingState="Naked")] expresses the first of these three dependencies.

The advantage of these data invariants should now be evident. Method TypeCheck can rely on the fact that the binding field of Identifier objects is non-null. Similarly, the back-end can rely on the fact that the type field of expression objects is non-null.

2.1 The aliasing problem

A technical reason that has kept typestate out of mainstream languages is the problem of maintaining correct typestate information in the presence of aliasing. To appreciate this problem, consider a method StripTypes:

```
// Set all type fields of expressions to null.
void StripTypes(AstNode ast);
```

```
class Expression : AstNode {
  [NotNull(WhenEnclosingState="Typed")]
  Type type;
   ...
}

class Identifier : Expression {
  string name;

  [NotNull(WhenEnclosingState="Bound")]
  AstNode binding;
   ...
}

class UnaryExpr : Expression {
  Operator oper;

  [NotNull(WhenEnclosingState="Naked,Bound,Typed")]
  [InState("Naked", WhenEnclosingState="Naked")]
  [InState("Bound", WhenEnclosingState="Bound")]
  [InState("Typed", WhenEnclosingState="Typed")]
  Expression operand;

   ...
}
```

Fig. 2. Typestate invariants of some front-end classes

This method sets all type fields of Expression objects back to null. The problem with this method is that the typestate of abstract syntax trees is weaker on exit than it is on entry. Thus, other pointers to nodes of the same abstract syntax tree may need to have their typestate weakened. Suppose for example that after type checking we build control-flow graphs (CFGs) that internally keep references to AstNodes satisfying typestate "Typed".

```
ast.TypeCheck(emptyTypeEnvironment);
CFG cfg= BuildCFG(ast);
StripTypes(ast);
WorkOnCFG(cfg);
```

The above code sequence is problematic, since the AstNode references in the cfg object are annotated to satisfy typestate "Typed", but after the call to StripTypes, these references point to objects that do not satisfy the typestate "Typed". In order to correctly track such non-local and non-monotonic typestate changes, strict non-aliasing regimes must be followed. The earliest attempts at typestate checking appear in a language called Nil that completely rules out aliasing in pointer structures [9]. Vault is a more recent programming language that permits typestate checking and strong aliasing control [3]. However, once an object's aliases are no longer statically known in Vault, its typestate needs to be *frozen*, that is, it can no longer change, except temporarily [5].

Although some typestate protocols will always require aliasing control (for example, open/close protocols), we identify in this paper a class of *heap monotonic* typestates that do not require aliasing control. The idea behind heap monotonic typestates is that, once a certain typestate is reached, no future changes to the

object will ever invalidate that typestate. Monotonic typestate checking makes it possible to capture object references in arbitrary typestates without the need to invalidate such references on future object updates. In our proposal, the method StripTypes cannot be typestate checked, since the update of the type field to null violates the monotonicity of our typestates.

3 Typestate formalization

We start with a number of definitions. Let Σ be a set of identifiers used to represent local variables and fields of objects. We use $\sigma \in \Sigma^+$ to represent access paths. Let V be the domain of values, including locations L used during program execution. A heap H is a map $L \times \Sigma \to V$, mapping location-field pairs to values. The local variables are accessed via a distinguished location containing all locals ℓ_{locals}. For convenience, define $H(\sigma) = H(\ell_{\text{locals}}, \sigma)$ and $H(\ell, x.\sigma') = H(H(\ell, x), \sigma')$.

Definition 1 (Heap monotonic predicate). *A predicate P on values and heaps is* heap monotonic*, if $P(v, H) \Rightarrow P(v, H')$ for every value v and heaps H and H', where H' is obtained from H by updates allowed by the static program semantics.*

Examples of heap monotonic predicates in most programming languages are notnull(v), null(v), dynamictype(v) $\leq T$, $v \geq 5$, etc.[1] Let $\mathcal{A}$ be a fixed set of heap monotonic predicates.

In a non-object-oriented setting, a typestate for an object o of type T is simply a named predicate over the fields of o. We write A_T for such a typestate predicate, where A is the name of the state. In this paper, we are interested in heap monotonic typestates. A typestate is by definition heap monotonic if it only depends on heap monotonic predicates. If field updates are restricted to preserve monotonicity (Sect. 3.4), then each heap monotonic typestate is itself a heap monotonic predicate. Thus the typestate of an object can depend on the typestates of objects stored in its fields.

3.1 Heap monotonic typestate in the presence of inheritance

In an object-oriented setting with single implementation inheritance, an object can be viewed as a list of *frames*, one per class in the inheritance path from the root class Object to its dynamic type.

We specify typestates of objects by giving a typestate per class frame. Thus, an object with dynamic type AstNode has a typestate for the Object frame and a separate typestate for the AstNode frame. A typestate A_T only describes fields declared in T, none in super or sub-classes of T.

We further need a technical device to abstract the typestate of sub-classes of an object, since the dynamic type of an object is rarely known statically. We thus

[1] These examples are trivially heap-monotonic, since they are independent of the heap.

introduce typestate predicates of the form $A_{\leq T}$. The meaning of this predicate is that A_S holds for every subclass S of T. Formally, if $\mathsf{supertype}(Q) = T$

$$A_{\leq T} \iff A_T \wedge A_{\leq Q}$$

Thus, the typestate annotations in our examples so far are interpreted as $A_{\leq \mathsf{Object}}$.

Let $\mathcal{MP}$ be the set of heap monotonic predicates consisting of $\mathcal{A}$ and all typestate predicates A_T. The interpretation of a typestate A_T is a map $[\![A_T]\!] \colon \Sigma \to 2^{\mathcal{MP}}$, mapping fields of T to the heap monotonic predicates that are true for the value contained in the field, when the enclosing object frame T satisfies state A. We interpret such sets of predicates as conjunctions.

Note that our typestates are not mutually exclusive. It is perfectly fine to have an $\mathsf{Expression}$ object in typestates $\mathsf{Bound}_{\mathsf{Expression}} \wedge \mathsf{Typed}_{\mathsf{Expression}}$. In fact, there is nothing in our typestate definitions that explicitly orders typestates. Since states are monotonic, "transitioning" an object from A_T "to" B_T results in an object with $A_T \wedge B_T$.

3.2 Language

We work with a small core object-oriented language consisting of classes, fields, and methods. Without loss of generality, we assume that each field uniquely identifies its declaring class. We describe statements modifying or accessing the heap and method calls, but omit other details. Typestate annotations are assumed to be given in the form of typestate predicates A_T discussed above, but we do not provide formal syntax and interpretation for such annotations here. The syntax used in the examples is one possible approach. We assume the language is statically typed, similar to C# or Java, and focus only on typestates.

method	$T.m(x_1, \ldots, x_n)$ returns z $\{\iota\}$
instruction sequence	$\iota ::= \cdot \mid s; \iota$
instruction	$s ::= x := y \mid x.f := y \mid y := x.f \mid y := \mathsf{null} \mid y := \mathsf{new}\, T()$
	$z := y_0.m(y_1, \ldots, y_n) \mid$
	$z := y_0.T.m(y_1, \ldots, y_n) \mid \cdots$

Instructions of interest consist of variable copy, field assignment, field read, null assignment, object construction, and virtual and direct method call.

3.3 Dynamic semantics

The semantics of the language is a standard small-step semantics of the form $(H, \iota) \to (H, \iota)$, relating machine states consisting of a heap H and instruction sequence ι. We omit the details for space reasons.

Let $[\![H]\!] : \Sigma^+ \to 2^{\mathcal{MP}}$ be the largest predicate assignment for heap H consistent with the rules in Fig. 3. We use the largest, rather than the inductively defined set in order to allow more properties of circular heap structures. Rule [H-atom] provides atomic heap-monotonic predicates. For example, for any location $\ell \neq \mathsf{null}$, $\mathsf{notnull}(\ell, H)$ holds. Rule [H-ts] relates predicates of fields $\sigma.f$ with

$\boxed{H \vdash \sigma : M}$

$$\frac{H(\sigma) = \ell \wedge p(\ell, H) \qquad p \in \mathcal{A}}{H \vdash \sigma : \{p\}} \text{ [H-atom]} \quad \frac{\forall f \in T. H \vdash \sigma.f : [\![A_T]\!](f)}{H \vdash \sigma : \{A_T\}} \text{ [H-ts]}$$

$$\frac{H \vdash \sigma : M_1 \wedge H \vdash \sigma : M_2}{H \vdash \sigma : M_1 \cup M_2} \text{ [H-and]} \qquad \frac{H(\sigma) = \mathsf{null}}{H \vdash \sigma : \{A_T\}} \text{ [H-ts-null]}$$

Fig. 3. Rules for deriving heap properties

the typestate of σ. Rule [H-ts-null] is a special case for null, which we consider to have every typestate.

$[\![H]\!]$ maps each access path σ to the observable heap monotonic predicates that hold for the value accessed through σ. Note that $[\![H]\!]$ is total, mapping access paths not in H to the empty set.

3.4 Static semantics

In this section, we formalize parts of the static semantics. We give type rules for the statements in our language. Of particular interest are the rules for interpreting and proving certain typestates, as well as the field update rule. We end by showing how to typestate check a small example.

Typestate checking cannot simply use a typestate environment (mapping identifiers to typestates) akin to a type environment, because we want to prove new typestates for a particular pointer stored at path σ, once sufficiently strong properties of its fields $\sigma.f$ are known. For this purpose, it is necessary to keep associations (akin to must-aliasing) that remember that a particular variable y holds the same value as $x.f$.

The static semantics thus uses two auxiliary structures, a heap abstraction $S\colon R \times \Sigma \to R$ and a predicate map $E\colon R \to 2^{\mathcal{MP}}$, where R is a finite set of symbolic pointers ρ. The heap abstraction S maps a symbolic pointer and a field to the symbolic pointer contained in that field. The predicate map E maps each symbolic pointer ρ to a set of predicates known to hold for ρ. Local variables are looked up as fields of a distinguished pointer ρ_{locals}. We use the short-hands $S(\sigma) = S(\rho_{\text{locals}}, \sigma)$ and $S(\rho, x.\sigma) = S(S(\rho, x), \sigma)$.

A symbolic pointer ρ abstracts an actual heap pointer in the following way. Each symbolic pointer corresponds to exactly one heap pointer. Multiple distinct symbolic pointers may correspond to the same heap pointer. For locals, the information is conservative must-alias information, that is, if $S(x) = S(y)$, then at runtime, $H(x) = H(y)$. The information for locals can be kept in synch with the actual execution, because in our language, locals are only assigned directly. For fields, the abstraction is more subtle. It doesn't correspond to must-aliasing exactly because we allow the abstraction to be outdated. For example, if $S(y) = S(x.f)$, then either $H(y) = H(x.f)$ for the current heap H or there is a past heap where the object referred to by x contains the current value of y in field

f, that is, a past $H' \leq H$, such that $H'(H(x), f) = H(y)$ This information is crucial, since it states that at some point in the past (heap H'), the must-aliasing information was correct. This allows us to deduce that if $H(x.f) \neq H(y)$, then there was an assignment to field f between heap H' and the current heap.

We now proceed to the typestate rules for each statement kind and observe how this static information is maintained and used. The static well-typestate relation for statements has the form $S, E \vdash \iota : S', E'$, where S and E are the static structures prior to the execution of statements ι, and S' and E' are the static structures after execution of ι.

Variable copy

$$\frac{S[(\rho_{\text{locals}}, x) \mapsto S(y)], E \vdash \iota : S', E'}{S, E \vdash x := y; \iota : S', E'}$$

The rule states that the remainder of the instructions ι are checked under the assumption $S(x) = S(y)$.

Field access

$$\frac{\begin{array}{l} S, E \vdash x.f : M \\ \rho \text{ fresh} \quad S(x) = \rho_x \\ S[(\rho_{\text{locals}}, y) \mapsto \rho][(\rho_x, f) \mapsto \rho], E[\rho \mapsto M] \vdash \iota : S', E' \end{array}}{S, E \vdash y := x.f; \iota : S', E'}$$

The result of reading field f is recorded under a fresh symbolic pointer ρ corresponding to the current pointer in $x.f$. We use a fresh pointer here, since the current static knowledge $S(x.f)$ may be outdated. But after the statement, we record the equality $S(x.f) = S(y)$, since it is definitely true at this point in the execution. The predicates M known to hold for $x.f$ prior to the statement are recorded in E for the fresh pointer ρ. The auxiliary judgment $S, E \vdash \sigma : M$ is used to deduce predicates for particular access paths. The rules are shown in Fig. 4.

For field accesses, the rules in Fig. 4 can prove properties in two ways: either by rule [TS-elim], applying knowledge of the typestate of x to the field $x.f$. Alternatively, by rule [Loc], where we use knowledge of the properties of the symbolic pointer $\rho = S(x.f)$ directly. It is not immediately obvious why this second way is sound, since we know that $x.f$ could be pointing to a new object, not corresponding to ρ. Fortunately, our rule for field update (shown later) enforces strong enough properties, that rule [Loc] can be proven sound. It is however necessary to restrict the knowledge of $E(\rho)$ to the set of predicates $\bigcup \mathcal{O}_{x.f}$ observable through path $x.f$. To see why, consider the following code snippet:

```
y := x.f;
y.EstablishStateA (); // establishes typestate A for y
// does x.f have typestate A?
```

$\boxed{S, E \vdash \sigma : M}$

$$\frac{E(S(x)) \supseteq M}{S, E \vdash x : M}\ [\text{Var}] \qquad \frac{S(x.f) = \rho \quad E(\rho) \cap \bigcup \mathcal{O}_{x.f} \supseteq M}{S, E \vdash x.f : M}\ [\text{Loc}]$$

$$\frac{S, E \vdash \sigma : M_1 \qquad S, E \vdash \sigma : M_2}{S, E \vdash \sigma : M_1 \cup M_2}\ [\text{Union}] \qquad \frac{}{S, E \vdash \sigma : \emptyset}\ [\text{Empty}]$$

$$\frac{S, E \vdash \sigma : \{A_T\} \qquad [\![A_T]\!](f) \supseteq M}{S, E \vdash \sigma.f : M}\ [\text{TS-elim}] \qquad \frac{\forall f \in T\,.\, S, E \vdash \sigma.f : [\![A_T]\!](f)}{S, E \vdash \sigma : \{A_T\}}\ [\text{TS-intro}]$$

Fig. 4. Rules for proving predicates of access paths

$\boxed{\mathcal{O}_\sigma}$

$$\mathcal{O}_x = \{\mathcal{MP}\}$$
$$\mathcal{O}_{\sigma.f} = \{[\![A_T]\!](f) \mid A_T \in \bigcup \mathcal{O}_\sigma\} \qquad f \in T$$

Fig. 5. Observable predicates for a given access path

We have to consider that $x.f$, after the call to EstablishStateA, may differ from y. (We make no assumptions about what is or is not modified.) There are two cases to consider. If $\bigcup \mathcal{O}_{x.f} \ni A_T$, then there is some typestate B_U of class U declaring field f such that $[\![B_U]\!](f) \ni A_T$ and we can conclude that $x.f$ satisfies typestate A, since—as we will see below—any update to $x.f$ during the call to EstablishStateA must have updated the field with an object satisfying state A_T. Otherwise ($\bigcup \mathcal{O}_{x.f} \not\ni A_T$), we cannot conclude $x.f$ has typestate A, since an update could have stored an object in $x.f$ not satisfying state A. As an example, consider field UnaryExpr.operand where $\mathcal{O}_{x.\mathsf{operand}} = \{\{\mathsf{notnull}, \mathsf{Naked}_{\leq \mathsf{Object}}\}, \{\mathsf{notnull}, \mathsf{Bound}_{\leq \mathsf{Object}}\}, \{\mathsf{notnull}, \mathsf{Typed}_{\leq \mathsf{Object}}\}\}$, therefore

$$\bigcup \mathcal{O}_{x.\mathsf{operand}} = \{\mathsf{notnull}, \mathsf{Naked}_{\leq \mathsf{Object}}, \mathsf{Bound}_{\leq \mathsf{Object}}, \mathsf{Typed}_{\leq \mathsf{Object}}\}$$

Field update

$$\frac{\begin{array}{l} S, E \vdash y : M \qquad M \supseteq \mathcal{U}_f(x) \\ S(x) = \rho_x \\ S' = S[(\rho_x, f) \mapsto S(y)] \\ S', E \vdash \iota : S'', E'' \end{array}}{S, E \vdash x.f := y; \iota : S'', E''}$$

The field update rule is the most crucial piece in our approach. We must first find an upper bound $\mathcal{U}_f(x)$ on the observable predicates of field $x.f$. This upper bound must account for all predicates of field f that any other access path to $x.f$

may already know or may be establishing. We will return below to our actual definition of $\mathcal{U}_f(x)$.

The field update is safe, if the condition $M \supseteq \mathcal{U}_f(x)$ is satisfied. This condition ensures that the update does not invalidate the typestate assumptions of any other access path. The static heap approximation is updated to reflect that at this point, $S'(x.f) = S'(y)$.

Let us now define $\mathcal{U}_f(x)$ to be the set

$$\bigcup\{M \mid \exists\sigma.\sigma \neq x \wedge H(\sigma) = H(x) \wedge (M \in \mathcal{O}_{\sigma.f}) \wedge ([\![H(x.f)]\!] \cup M \text{ consistent})\}$$

Note that we formulate $\mathcal{U}_f(x)$ in terms of knowledge of the dynamic heap H that in general won't be statically known. In the absence of any aliasing information on x or knowledge about the value $x.f$, the bound goes up to $\bigcup \mathcal{O}_{z.f}$. In this case, field updates require that the written value satisfy all predicates that could ever be observed of field f. This worst estimate allows each field to go only from the null-initialized state (as established on entry to the constructor), to the fully initialized state.

Another extreme case is when we know that x is the only pointer to the object whose field is updated (depending on the programming language, for example during or right after construction). In that case, $\mathcal{U}_f(x) = \emptyset$, and any update is valid (even a non-monotonic one).

Another possible case is when we know something about the current value of $x.f$. In general, we need to include in $\mathcal{U}_f$ only predicate sets M that are consistent with the current properties of the field $[\![H(x.f)]\!]$. For example, if $x.f$ is null, we can compute $\mathcal{U}_f(x)$ to be the union of all observable predicates for f that are consistent with null. Consider field operand of class UnaryExpr in Fig. 2. If we know that $x.f$ is null at the moment of the update, we can conclude that no pointer to the unary expression object can assume it in any of the typestates Naked,Bound,Typed, since they all include predicate notnull, which is inconsistent with the current value of $x.f$.

Methods A virtual method signature consists of pre and post predicates for each method parameter and result, including the receiver this. We refer to these predicates by $\mathsf{pre}(m.x)$ and $\mathsf{post}(m.x)$, where m is the name of a declared method, and x is the parameter name, this, or return. Annotations [Pre(A)] on x translate to $\mathsf{pre}(m.x) = A_{\leq \mathsf{Object}}$. Annotations [Post(B)] on x translate into $\mathsf{post}(m.x) = B_{\leq \mathsf{Object}}$.

A particular implementation of method m in a class T is referred to as $T.m$. The signature of $T.m$ differs with respect to the virtual method signature only in the treatment of the receiver post condition. If the post condition for the receiver in a declared virtual method is $A_{\leq \mathsf{Object}}$, then the post-condition for the receiver in a particular implementation method $T.m$ is weaker, namely

$$\mathsf{post}(T.m.\mathsf{this}) = \bigwedge_{S \geq T} A_S$$

that is, the conjunction of all typestates A_S for class frames S at or above T. The frames of strict subclasses of T obtain no stronger properties.

It should intuitively be clear why this is so. A method implementation $T.m$ can only directly affect the state of the object at or above class frame T. To produce a deep post condition of the form $A_{\leq \mathsf{Object}}$, a virtual dispatch is needed.

We now give the conditions for typestate checking of a method body. Assume $x_0 = \mathsf{this}$.

$$\frac{\begin{array}{ll} S = [(\rho_{\text{locals}}, x_i) \mapsto \rho_i] & i = 0..n, \rho_i \text{ fresh} \\ E = [\rho_i \mapsto \mathsf{pre}(T.m.x_i)] & i = 0..n \\ S, E \vdash \iota : S', E' & \\ S', E' \vdash x_i : \mathsf{post}(T.m.x_i) & i = 0..n \\ S', E' \vdash y : \mathsf{post}(T.m.\mathsf{return}) & \end{array}}{\vdash T.m(x_1, \ldots, x_n) \text{ returns } y \ \{\iota\}}$$

The first two lines describe the initial environment S, E in which the method body is typed. We assume a distinct symbolic pointer ρ_i for each parameter, and populate E with the respective preconditions. Note that we need not have any knowledge about possible aliasing of the parameters. The third line checks the body ι, resulting in the static structures S' and E' describing the typestates at exit of the method. Finally, the last two lines check the post-conditions of the parameters, this, and the result.

We assume that each class T implements each virtual method of any parent class. If no implementation is explicitly given, the implementation

{ **return this**.**base**.m(x_1,...,x_n); }

is assumed, where base is the immediate supertype of T. This requirement is necessary to check the correctness of any specified typestate changes on the receiver this.

Method calls We allow both virtual calls and non-virtual (direct) method calls. The two differ only in whether the virtual method signature or a particular implementation signature is used. We give the rule for virtual calls. Direct calls look identical, but every occurrence of m is replaced with $T.m$. Assume $x_0 = \mathsf{this}$.

$$\frac{\begin{array}{ll} S, E \vdash y_i : \mathsf{pre}(m.x_i) & i = 0..n \\ E' = E[S(y_i) \mapsto E(S(y_i)) \cup \mathsf{post}(m.x_i)] & i = 0..n \\ S' = S[(\rho_{\text{locals}}, z) \mapsto \rho] & \rho \text{ fresh} \\ E'' = E'[\rho \mapsto \mathsf{post}(m.\mathsf{return})] & \\ S', E'' \vdash \iota : S''', E''' & \end{array}}{S, E \vdash z := y_0.m(y_1, \ldots, y_n); \iota : S''', E'''}$$

The first line ensures that the arguments in the calling context satisfy the preconditions of the virtual method parameters. The second line joins the post typestate of each parameter to the current knowledge in the predicate map. The third line adds a fresh symbolic pointer ρ for z to the store abstraction. The fourth line adds the result typestate for ρ to the predicate map. The last line ensures that the static environment right after the method call is sufficient to prove typestate safety of the remaining instruction sequence ι.

3.5 Example revisited

We now return to our motivating example and show the TypeCheck method for unary expressions.

```
[Pre("Bound"),Post("Typed")]
UnaryExpr.TypeCheck()
{
  y := this.operand;
  y.TypeCheck();
  this.type := new Type(...);
}
```

The initial environment is as follows: $S(\mathsf{this}) = \rho_0, E(\rho_0) = \{\mathsf{notnull}, \mathsf{Bound}_{\leq \mathsf{Object}}\}$. After the assignment to y, we also have $S(y) = \rho_1, S(\rho_0, \mathsf{operand}) = \rho_1, E(\rho_1) = [\![\mathsf{Bound}_{\mathsf{UnaryExp}}]\!](\mathsf{operand}) = \{\mathsf{notnull}, \mathsf{Bound}_{\leq \mathsf{Object}}\}$. Our static information satisfies the precondition to invoke TypeCheck on y. On return from the call, the environment is updated to $E(\rho_1) = \{\mathsf{notnull}, \mathsf{Bound}_{\leq \mathsf{Object}}, \mathsf{Typed}_{\leq \mathsf{Object}}\}$. After the assignment to this.type, the environment is updated to $S(\rho_0, \mathsf{type}) = \rho_2, E(\rho_2) = \{\mathsf{notnull}\}$. Call this environment S', E'. At this point, we can prove the post condition on the receiver: $\mathsf{Typed}_{\mathsf{Object}} \wedge \mathsf{Typed}_{\mathsf{AstNode}} \wedge \mathsf{Typed}_{\mathsf{Expression}} \wedge \mathsf{Typed}_{\mathsf{UnaryExpr}}$. The first two predicates in the conjunct are trivial, since their typestate mapping is empty. So for each field f of Object and AstNode, we can prove $S', E' \vdash \mathsf{this}.f : \emptyset$, and then conclude via rule [TS-intro] (twice) and [TS-union] that $S', E' \vdash \mathsf{this} : \{\mathsf{Typed}_{\mathsf{Object}}, \mathsf{Typed}_{\mathsf{AstNode}}\}$. For $\mathsf{Typed}_{\mathsf{Expression}}$, we need to prove $S', E' \vdash \mathsf{this.type} : \{\mathsf{notnull}\}$, which we do via rule [Loc]. Similarly, we prove $S', E' \vdash \mathsf{this.operand} : \{\mathsf{notnull}, \mathsf{Typed}_{\leq \mathsf{Object}}\}$ via [Loc].

3.6 Soundness

The soundness of the system is subtle because it relies on the field update guarantees and the use of out of date field information. We have proven soundness of the hard cases (field update and method call) via a standard subject reduction approach [11].

4 Discussion

This section discusses some additional properties of heap monotonic typestates and considers possible extensions.

4.1 DAGs and circular structures

Our typestate proposal works for arbitrary graph structures, not just trees or DAGs. Establishing arbitary typestate relations among DAG nodes is done by traversing the DAG as if it were a tree. To avoid the duplicate traversals, dynamic typestate tests (see next subsection) can be used.

Surprisingly, the static proof technique described here can prove typestate properties of circular structures as well. Consider a general graph, where each node satisfies typestate A_N, if some internal field f is non-null, and all successor nodes are in A_N. The general way to establish that each node in an arbitrary graph satisfies A_N is to first build a corresponding DAG, where the missing back pointers point to a dummy object for which it is trivial to establish typestate A_N. It is then possible to prove inductively that each node satisfies A_N. After that, do one more traversal, where all pointers going to the dummy node are updated to point to their intended node. Since their intended target is already in typestate A_N, the update is okay.

Since the approach requires updating of back pointers, it can only be used to establish the ultimate typestate of each graph node. It seems not possible to gradually transition the entire circular graph to better typestates as in our AST example.

4.2 Dynamic typestate tests

Our approach can prove properties of a DAG inductively, simply by traversing it as a tree. However, shared subtrees have to be traversed multiple times. It would be desirable to dynamically record knowledge of an established typestate in a field, and allow a dynamic test of such a field to infer the entire typestate of the object.

Let's call such a field a typestate designator and mark these fields specially with an annotation [TypeState]. Further assume, that such fields must be of type string, and that we have the global invariant that for every object o, if $o.f$ is a typestate designator, and $o.f$ is equal to "A", then o satisfies $A_{\leq \mathsf{Object}}$.

Writing to a typestate designator requires a compile-time known string "A", and a proof that the enclosing object satisfies $A_{\leq \mathsf{Object}}$. Furthermore, in the true branch of a conditional test of the form if (o.f == "A"), the typestate of o can be assumed to be $A_{\leq \mathsf{Object}}$.

Using this device, DAG traversals can establish typestate properties without visiting nodes more than once.

4.3 Relational atomic predicates

So far, all our heap monotonic atomic predicates are unary, that is, they involve exactly one value of a single field. An obvious generalization is to allow relational predicates, such as x≤y, where x and y are both fields of the same class frame. Updates to such fields are slightly more tricky, since to maintain the relational property, two updates may be required, and the property may not hold after the first update. As long as one can statically prove that both updates occur atomically, such extensions can be handled.

The requirement that both fields are of the same class frame is to maintain modular soundness. Without this requirement, modifications to a field of class T, may invalidate the invariant of a subclass S of T, but the code in class T has no knowledge of such an invariant.

4.4 Concurrency

Our typestate checking remains sound in the presence of concurrency. That static rules indeed assume that after each instruction, every field could be updated by another thread (provided the update satisfies our field update rule).

5 Related work

The extended static checker (ESC) project uses theorem proving to enforce method-level specifications and object invariants [4]. The enforcement of object invariants however is sound only under strict non-aliasing conditions. The work on Vault [3] and Roles [8] soundly enforce object invariants and allow state changes. However, both systems require non-aliasing assumptions. Earlier work on alias types also allows incrementally establishing data structure invariants, but only under non-aliasing assumptions [10].

The present system has the advantage that it is useful even without any non-aliasing assumptions, but it can exploit non-aliasing assumptions to allow non-monotonic typestate changes.

The work on type checking Java byte code for safety properties by Freund and Mitchell considers only whether the constructor of a newly allocated object is called before other methods [7]. It does not enforce field initializations or other object invariants.

The motivation for the present work stems in part from our prior work on guaranteeing initialization of object fields to non-null pointers [6], and also from typestate-like code comments we found in a front-end written by Dave Hanson.

6 Conclusion

Initialization of objects often happens gradually over the lifetime of an object, rather than during execution of the constructor alone. We believe that monotonically evolving typestate provides a good match for capturing such evolving object invariants. It has the advantage over prior work that it requires no non-aliasing guarantees, but can exploit them if they are present. The resulting programming model seems flexible.

References

1. Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 211–230, November 2002.
2. David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA'98 Conference Proceedings*, pages 48–64, October 1998.

3. Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 59–69, June 2001.
4. David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, 1998.
5. Manuel Fähndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–24, June 2002.
6. Manuel Fähndrich and K. Rustan M. Leino. Non-null types in an object-oriented language, 2002. Presented at the 2002 Workshop on Formal Techniques for Java-like Languages.
7. Stephen N. Freund and John C. Mitchell. The type system for object initialization in the Java bytecode language. *ACM Transactions on Programming Languages and Systems*, 21(6):1196–1250, 1999.
8. Viktor Kuncak, Patrick Lam, and Martin Rinard. Role analysis. In *Conference Record of the 29th Annual ACM Symposium on Principles of Programming Languages*, 2002.
9. Robert. E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE TSE*, 12(1):157–171, January 1986.
10. David Walker and Greg Morrisett. Alias types for recursive data structures. In *Proceedings of the 4th Workshop on Types in Compilation*, September 2000.
11. Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994.