Typestates for Objects

Robert DeLine and Manuel Fähndrich

Microsoft Research
One Microsoft Way
Redmond, WA 98052-6399 USA
{rdeline,maf}@microsoft.com

Abstract. Today's mainstream object-oriented compilers and tools do not support declaring and statically checking simple pre- and postconditions on methods and invariants on object representations. The main technical problem preventing static verification is reasoning about the sharing relationships among objects as well as where object invariants should hold. We have developed a programming model of typestates for objects with a sound modular checking algorithm. The programming model handles typical aspects of object-oriented programs such as downcasting, virtual dispatch, direct calls, and subclassing. The model also permits subclasses to extend the interpretation of typestates and to introduce additional typestates. We handle aliasing by adapting our previous work on practical linear types developed in the context of the Vault system. We have implemented these ideas in a tool called Fugue for specifying and checking typestates on Microsoft .NET-based programs.

1 Introduction

Although mainstream object-oriented languages, like C♯ and Java, automatically catch or prevent many programming errors through compile-time checks and automatic memory management, there remain two related sources of error that often manifest as runtime exceptions. First, a developer must obey the rules for properly calling an object's methods, including calling them in an allowed order and obeying the preconditions on the methods' arguments. Today, such rules are captured in the class's documentation, if at all. Second, a developer implementing a class must ensure that each public method upholds the class's representation invariant, including any invariants inherited from the superclass.

Types are the main mechanism through which programmers currently specify mechanically checked preconditions, postconditions and representation invariants. These mechanical checks are critical for spotting errors early in the development cycle, when they are cheapest to fix. Types, however, are a very limited specification tool, particularly in imperative programming languages, where objects change state over time. Using standard type systems, we cannot address the sources of errors described above. On the other hand, providing the programmer with a rich logic for writing preconditions, postconditions, and

object invariants quickly runs into decidability problems. For example, the ESC/Java system [1] supports rich specifications, but does not fully verify object invariants.

In this paper, we propose a statically checkable *typestate* system to declare and verify state transitions and invariants in imperative object-oriented programs. Typestates [2] 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's state changes, its typestate may change as well. Typestates provide an abstraction mechanism for predicates over object graphs, but retain some of the simplicity and feel of types. Typestates can be used to restrict valid parameters, return values, or field values, and thereby provide extra guarantees on internal object properties.

Previous research demonstrated the utility of typestates for capturing interface rules in non-object-oriented, imperative languages [2, 3]. In this paper, we adapt and extend the programming methodology [3, 4] that we developed for reasoning about non-object oriented imperative programs to the object-oriented setting. The technical contributions are the following:

Typestates are modular. A typestate is a description of the contents of all of an object's fields. However, at a given program point, a modular static checker only knows about those fields declared in an object's declared type or its superclasses. Hence, some of the object's state (introduced by subclasses) is unknown. Typestates are a good match for this problem, since typestates provide names for abstract predicates over field state. A modular static checker can know an object's state by name without knowing the exact invariant that an unknown subclass associates with that name. This approach allows a clean description of how subclasses can extend the interpretation of a typestate and the language features of upcasts, downcasts, and virtual method invocation. We also describe the limits of expressiveness of our typestate formulation.

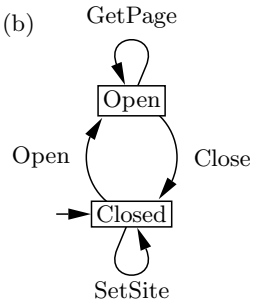
Typestates generalize object invariants. Commonly, an object invariant (or representation invariant) is a predicate that is established during object construction that remains true throughout the object's lifetime. We believe in practice that this view is too limiting, since objects tend to satisfy different properties at different stages of their lifetimes. Instead, we view an object as having different typestates over its lifetime, where each typestate is a named predicate over the object's concrete state. By making the object typestate explicit at pre- and postconditions of all methods, we also avoid the problem of defining where object invariants must hold, which in the past has been approached using ad-hoc notions of "visible states" [5]. In our model, traditional object invariants are simply properties that are common to all typestates of the object.

Typestates support incremental object state changes. Any given method implementation only has a partial view of an object. Hence, describing how an object's state (including the statically unknown subclass state) changes is nontrivial. Furthermore, because an object's state change is necessarily im-

(a)

```
[ TypeStates("Open", "Closed") ]
class WebPageFetcher {

  [ Post("Closed"), NotAliased ]
  WebPageFetcher([NotNull] string s)
  {
    this.site = s;
  }

  [ Pre("Closed"), Post("Open"), NotAliased ]
  virtual void Open()
  {
    this.cxn = new Socket();
    this.cxn.Bind(this.site);
    this.cxn.Connect();
  }

  [ Pre("Open"), Post("Closed"), NotAliased ]
  virtual void Close() {
    this.cxn.Close();
    this.cxn = null;
  }

  [ Pre("Open") ]
  [ return: NotNull ]
  virtual string GetPage( [NotNull] string path) {
    return this.cxn.Receive();
  }

  [ Pre("Closed") ]
  virtual void SetSite ([NotNull] string site) {
    this.cxn = site;
  }

  [ NotNull(WhenEnclosingState="Open") ]
  [ NotAliased(WhenEnclosingState="Open") ]
  [ InState("Connected", WhenEnclosingState="Open") ]
  [ Null(WhenEnclosingState="Closed") ]
  private Socket cxn;

  [ NotNull ]
  private string site;
}
```

(b)



(c)

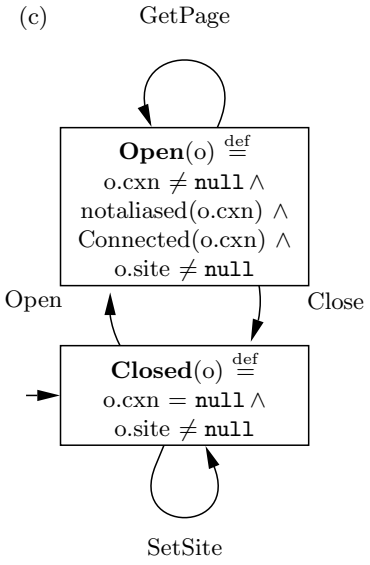


Fig. 1. Web page fetcher example

plemented incrementally (by changing individual fields), typestates must be able to describe intermediate states, where different parts of an object have different states. We introduce *frame typestates* and *sliding method signatures* to address these issues.

2 Motivating example

This section informally introduces typestates for objects and illustrates some of its expressive power, as well as technical aspects that we will further discuss in the rest of the paper. In our examples, we use C♯ syntax and attributes (in brackets) to record typestates, pre-, and postconditions. Later, in Section 6, we

```
[ TypeStates("Raw", "Bound", "Connected", "Closed") ]
class Socket {

  [ Post("Raw"), NotAliased ]
  Socket();

  [ Pre("Raw"), Post("Bound"), NotAliased ]
  void Bind(string endpoint);

  [ Pre("Bound"), Post("Connected"), NotAliased ]
  void Connect();

  [ Pre("Connected") ]
  void Send(string data);

  [ Pre("Connected") ]
  string Receive();

  [ Pre("Connected"), Post("Closed"), NotAliased ]
  void Close();
}
```

Fig. 2. Simplified socket interface

will introduce a small formal language to make these examples precise. Fig. 1(a) contains the source of a simple class WebPageFetcher providing the functionality to open a particular web server, to fetch pages from the server, and to close the connection to the server. A WebPageFetcher object can be in one of two typestates, Open and Closed. The constructor produces an object in typestate Closed, the method Open changes the object's typestate from Closed to Open, and the method Close changes the typestate back from Open to Closed. Method GetPage can be called only when the object satisfies typestate Open, but does not change the typestate. Similarly, method SetSite can only be called when the object satisfies typestate Closed. These state changes can be pictured as the finite state machine in Fig. 1(b).

The Pre and Post annotations on methods restrict the order of operations that clients can invoke on the object. Such order restrictions are useful because a method's implementation makes assumptions about the object's state when it is invoked. For example, calling GetPage on a Closed object results in a null dereference exception because the method's code assumes that the field cxn is not null. In our approach, we make the relationship between typestate and object invariants explicit.

The annotations on the fields cxn and site define each typestate in terms of what properties of the object's concrete state hold in that typestate. If the object is in state Closed, the private Socket cxn to the web server is null. If the object is in state Open, then the private Socket cxn is non-null and in typestate Connected, which is a typestate of the Socket class, as shown in Fig. 2. The annotation NotNull on field site specifies a classic invariant, since it is not qualified by a particular typestate and therefore holds in all typestates.

o	Closed(o)	Open(o)
WebPageFetcher	o.cxn = null ∧ o.site ≠ null	o.cxn ≠ null ∧ notaliased(o.cxn) ∧ Connected(o.cxn) ∧ o.site ≠ null

Fig. 3. Typestate interpretation

In short, a typestate is a predicate over an object's field state. To a client, this predicate is an uninterpreted function to be matched by name; to an implementor, the predicate is defined in terms of predicates over the fields' values. Fig. 1(c) shows the same state machine as in (b), with each state enlarged to show the predicate that holds in that typestate. The state machine in (b) is the client's view of a WebPageFetcher object, while the state machine in (c) is the implementor's view.

Given these annotations, we can mechanically verify that every method implementation assumes only the stated precondition and guarantees the stated postcondition. Starting with the constructor, we observe that it sets field site to the non-null constructor argument s. That satisfies the invariant of field site in typestate Closed. However, field cxn is not assigned. In our approach, we assume that the implicit pre-state of objects in constructors is typestate Zeroed, in which all fields are initialized to their zero-equivalent value. In state Zeroed, field cxn contains null and therefore satisfies the necessary condition for typestate Closed. Since both of the WebPageFetcher's fields satisfy the conditions for typestate Closed at the end of the constructor, the constructor satisfies its postcondition.

Method Open is interesting in that it changes the typestate of the receiver from Closed to Open. It does so by initializing field cxn to a fresh socket. After the constructor call, the socket has typestate Raw (see Fig. 2). Thus, before satisfying the postcondition of method Open, the socket has to be put into the right typestate by calling methods Bind and Connect in that order, according to the Pre and Post annotations in class Socket. After calling Bind and Connect, the field cxn is in typestate Connected and the field site is unchanged (and therefore still non-null). Hence, the receiver is in typestate Open and, the method Open satisfies its postcondition.

So far, we have ignored two issues: 1) the annotations NotAliased appearing in the code, and 2) the fact that there could be subclasses of WebPageFetcher. The next two sections deal with object typestates and subclasses. Section 5 describes a programming model that allows static tracking of typestates in the presence of aliasing.

3 Object typestate

Consider again the typestates of our WebPageFetcher example. We can view these in the form of the table in Figure 3. The table maps a class and typestate to a

formula over the fields of an object of that class. The formula consists of atomic predicates such as value equalities, aliasing assumptions, as well as recursive typestate assumptions about the state of objects referenced through fields. In this paper we assume that formulae include at least equalities and disequalities between variables and null. In practice, any richer theory for which there are decidable satisfiability checkers is suitable.

As we can see from this table, the typestates we have used so far in our examples were really not typestates of an entire object, but only *frame typestates*, *i.e.*, a typestate of a particular *class frame* of an object. A class frame of an object is the set of fields of the object declared in that particular class, not in any super- or sub classes. In our example so far, we used frame typestates expressing properties of the WebPageFetcher frame.

To obtain an *object typestate*, we must be able to describe properties of all frames of an object, leading to the following issues:

Modularity of the typestate definition The meaning of an object typestate cannot be fully defined when the typestate is introduced, subclasses must be able to give new interpretations of typestates for their own fields. Nevertheless, a typestate should describe all parts of an object, even unknown subclasses. We address these issues in Section 3.1.

Non-uniformity of typestates Because a change in typestate is implemented as individual field updates, an object's typestate changes only gradually. Hence, typestates must be able to describe intermediate states of objects, where part of the object is in state A, other parts in state B. Section 4 discusses this issue in more detail.

To accommodate the above problems we define an object typestate to be a collection of frame typestates, one per class frame of the object's allocated (dynamic) type. In other words, we use one frame typestate for the static type frame of a reference, one for each super class frame, and one for each potential subclass frame. Since we are interested in a modular system, we cannot statically know the subclasses of a particular type. Therefore, our object typestates require an abstraction that gives a *uniform* typestate to all unknown subclass frames. Formally, an object typestate takes the form σ_C:

$$
\begin{array}{lll}
\text{(object typestate)} & \sigma_C ::= \chi_C :: \mathbf{rest}@s \mid \chi_C :: \bullet & \\
\text{(frames)} & \chi_C ::= \chi_B :: C@s & \text{where } B = \mathsf{baseclass}(C) \\
& \quad \mid \mathsf{Object}@s & \text{where } C = \mathsf{Object}
\end{array}
$$

An object typestate σ_C is a collection of frame typestates χ_C and either a rest typestate s (specifying that all possible subclass frames of C satisfy s), or no rest state indicating that the dynamic object type is exactly C (for example, right after new, or because class C is restricted from having subclasses, as with sealed classes in C$^\sharp$). A collection of frame typestates χ_C consists of a frame typestate for frame C and each supertype of C. Class C in an object typestate σ_C corresponds to the traditional static type of an object.

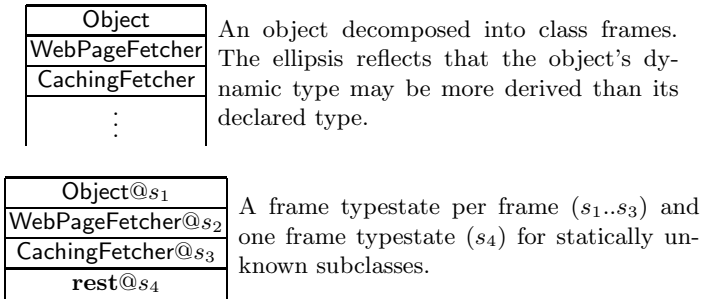


Fig. 4. Illustration of class frames and frame typestates

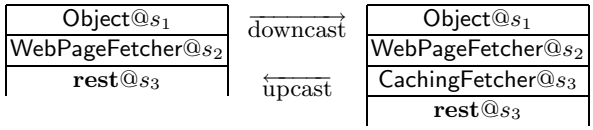


Fig. 5. Illustration of upcast and downcast with typestates

To keep the presentation simple, our formulation assumes a single typestate per class frame whereas in principle, an object can satisfy multiple compatible typestates.

Fig. 4 graphically illustrates the idea of separate frame typestates in an object and the rest state for all subclasses. Our model contains the restriction that a frame typestate can only constrain fields in that frame, but not in any frames of superclasses. This restriction enables modular reasoning in that writing a field of a particular frame can only affect that frame's typestate, but none of the typestates of any subclass frames.

Fig. 5 illustrates how upcasts and downcasts work in the presence of typestates. To downcast to an immediate subclass, the newly materialized frame typestate is simply equal to the original rest typestate for the subclasses (s_3). For an upcast, the disappearing frame must have the same typestate as the rest state that absorbs the frame; otherwise, the upcast is illegal.

Typestate shorthands in examples Explicitly specifying each frame typestate of an object in our examples is a nuisance. We use the following convention to specify entire object typestates. By default, a typestate specification applies to the class frame that introduces the typestate name (via the TypeStates annotation on the class) and to all subclass frames. Alternatively, each typestate annotation can be targeted at a particular frame C using the qualifier Type=C, or at the frames of all subclasses using the qualifier Type=Subclasses in Pre, Post, or InState annotations. The root class Object has a single typestate Default, which all

classes inherit. Those frames for which no explicit typestate is given are assumed to be in typestate Default.

Given this description, we can now interpret the object typestates specified in class WebPageFetcher. Method WebPageFetcher::Open, for instance, specifies the receiver precondition object typestate Object@Default :: WebPageFetcher@Closed :: **rest**@Closed, and method WebPageFetcher.GetPage specifies the receiver precondition Object@Default :: WebPageFetcher@Open :: **rest**@Open.

3.1 Typestates and subclasses

Let us now turn to the subclass CachingFetcher in Fig. 6 to see how typestates work in the presence of object extensions. The caching web page fetcher has a new cache field to hold a cache of already fetched pages. The natural invariants for this field are that it is null when the fetcher is Closed and non-null when the fetcher is Open. Since the subclass can provide it's own interpretation for typestates Open and Closed, adding this invariant is not a problem. The following table summarizes the frame typestates for a CachingFetcher object.

o	Default(o)	Closed(o)	Open(o)	CacheOnly(o)
Object	true	Default(o)	Default(o)	Default(o)
WebPageFetcher	o.cxn $\neq$ null$\wedge$ maybealiased(o.cxn)$\wedge$ Default(o.cxn)$\wedge$ o.site $\neq$ null	o.cxn = null$\wedge$ o.site $\neq$ null	o.cxn $\neq$ null$\wedge$ notaliased(o.cxn) $\wedge$ Connected(o.cxn)$\wedge$ o.site $\neq$ null	Default(o)
CachingFetcher	o.cache $\neq$ null	o.cache = null	o.cache $\neq$ null	o.cache $\neq$ null

This table illustrates the two ways in which a subclass can extend its superclasses' typestates: (1) by associating its own field invariants to a typestate defined in a superclass (e.g., the invariants on field cache); and (2) by adding new typestates (e.g., the typestate CacheOnly). Notice that when a typestate is not defined for a given frame, we use the typestate Default.

Looking at method override CachingFetcher.GetPage, we see how this method can take advantage of the object typestate precondition on the receiver. Since the parent method WebPageFetcher.GetPage specified frame typestate Open for all subclass frames (an abstract specification, since these typestate have not been defined at that point), the overriding method can rely on the extra properties provided by its frame typestate. If we had instead limited the typestate of subclasses to a known predicate (e.g., true), subclasses could not make any assumptions about their own fields in such overridden methods. In our example, CachingFetcher.GetPage assumes that field cache is not null.

4 Sliding methods

Given the ideas presented so far, there is a problem with methods that purport to change the typestates of subclass frames, such as WebPageFetcher::Open. Its Post specification states that all frames, including all unknown subclasses, will be in typestate Open at the end of the method body. As written, this specification is unfortunately not satisfied by the implementation. The method can assign only

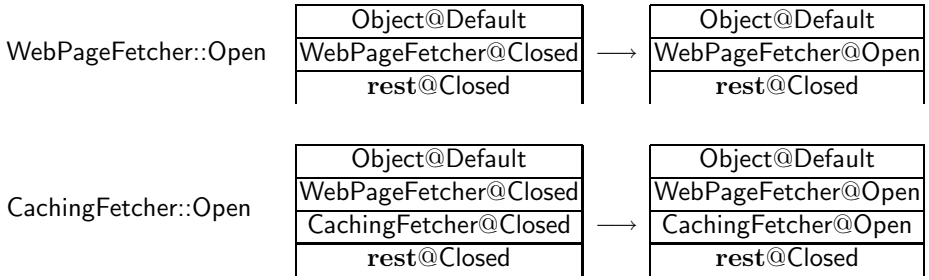
```
[ TypeStates("CacheOnly") ]
class CachingFetcher : WebPageFetcher {

  [ Post("Closed"), NotAliased ]
  CachingFetcher(string site ) : base( site ) {}

  [ Pre("Closed"), Post("Open"), NotAliased ]
  override void Open()
  {
    base.Open();
    this.cache = new Hashtable();
  }

  [ Pre("Open"), Post("Closed"), NotAliased ]
  override void Close()
  {
    base.Close();
    this.cache = null;
  }

  [ Pre("Open") ]
  [ return: NotNull ]
  override string GetPage( [NotNull] string path)
  {
    string page = this.cache.GetValue(path);
    if (page == null) {
      page = base.GetPage(path);
      this.cache.Add(path, page);
    }
    return page;
  }

  [ Null(WhenEnclosingState="Closed"), NotNull(WhenEnclosingState="Open,CacheOnly") ]
  private Hashtable cache;

  [ Pre("Open"), Post("CacheOnly"), Post("Closed", Type=WebPageFetcher), NotAliased ]
  void CloseKeepCache()
  {
    base.Close();
  }

  [ Pre("CacheOnly"), Pre("Closed", Type=WebPageFetcher) ]
  [ return:MayBeNull ]
  string GetCachedPage(string path)
  {
    string page = this.cache.GetValue(path);
    return page;
  }

  [ Pre("CacheOnly"), Pre("Closed", Type=WebPageFetcher), Post("Closed"), NotAliased ]
  void DeleteCache() {
    this.cache = null;
  }

}
```

Fig. 6. Caching web page fetcher

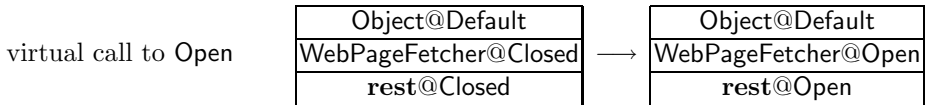
to fields that are visible through the static type and therefore cannot have any effect on fields of subclasses. Thus, on exit, the subclass typestates must still be Closed. This begs the question how a method can possibly change subclass states.

A method can only directly affect fields of its frame or the fields of superclasses. In order to change the typestate of subclasses, a virtual method call to a *sliding method* is required. The idea behind a sliding method is that each subclass implements a slightly stronger state change, namely each subclass changes the typestate of its frame and all frames of superclasses, but leaves the subclass typestates unchanged. As long as each subclass correctly implements such a sliding method, a virtual call to a sliding method is guaranteed to change the entire object typestate, since it dispatches to the dynamic type (the lowest class frame) and changes that frame as well as all super class frames.

We call such methods *sliding* because the typestate of the class introducing the method keeps sliding down the class hierarchy with each subtype, overriding the rest state. Graphically, we can illustrate this as follows:



In the limit, i.e., the effect observed for a virtual call, the signature is simply:



We fix our example by relaxing the post typestate of subclasses to remain Closed in methods Open and Close, and by treating these methods as sliding methods. For method Open in both classes WebPageFetcher and CachingFetcher, the corrected annotations are:

```
[ Pre("Closed"), Post("Open"), Post("Closed", Type=Subclasses) ]
virtual void Open () { ... }
```

4.1 Sliding signatures

In general, for each virtual method M, we thus have a family of related method signatures: the virtual signature $\mathsf{virtSig}_C$ (used at a virtual call site), and one implementation signature $\mathsf{implSig}_D$ per implementation of the method by class D. These signatures are derived from the implementation signature of M in class C which introduces virtual method M. The signature relations are illustrated in Figure 7 and formally defined in Figure 15. We assume that class C introduces sliding method M, and D is some subclass of C (notation $D \ll C$). The pre and post states of base classes of C are χ_B^{pre} and χ_B^{post}.

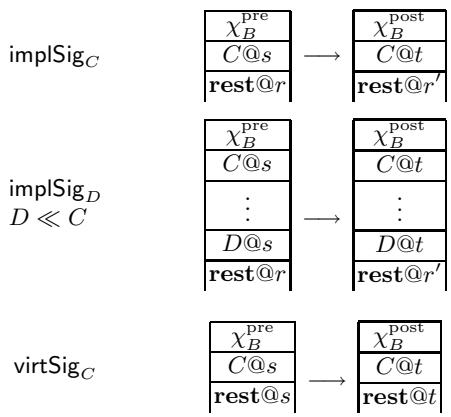


Fig. 7. Relation between typestates of this in signatures of sliding method implementations and the virtual call signature

Note that the rest states r and r' used in the implementation method signatures can be arbitrary (they need not be related to the pre-state s or the post-state t). It is instructive to study the different possible scenarios and what they imply for implementations.

For method Open, we have C = WebPageFetcher, D = CachingFetcher, s = Closed, t = Open, and $r = r'$ = Closed. This specification requires that the implementation of CachingFetcher::Open calls the base-class Open method *before* changing its own frame state, for otherwise, the pre-condition at the base-class call is not satisfied for its frame. An alternative is to pick $r = r'$ = Open, which forces implementations of Open to first change their own frame to state Open before calling the base-class method. Finally, if $r = r'$ is yet a third state, then implementations must first change their own frame to r, then call the base-class method, then change their own frame from r to Open.

In most signatures, rest states r and r' are equal. However, there are useful cases where that is not the case. Consider for instance a wrapper method containing a virtual call to Open. It would have s = Closed, r = Closed, t = Open, and r' = Open.

Given these definitions, we can now illustrate how to prove that method CachingFetcher::Open implements its signature correctly. Note how CachingFetcher::Open calls the base class Open method before changing its own frame. This base call is non-virtual, and therefore the method signature implSig$_{\text{WebPageFetcher}}$ applies (rather than the virtual signature). We thus have the progression of the receiver object through the following typestates:

- on entry (implSig$_{\text{CachingFetcher}}$)

  Object@Default :: WebPageFetcher@Closed :: CachingFetcher@Closed :: **rest**@Closed
- upcast for direct base-call to WebPageFetcher::Open

Object@Default :: WebPageFetcher@Closed :: **rest**@Closed

– direct base-call to WebPageFetcher::Open ($\mathsf{implSig}_{\mathsf{WebPageFetcher}}$)
Object@Default :: WebPageFetcher@Open :: **rest**@Closed
– safe down-cast to CachingFetcher
Object@Default :: WebPageFetcher@Open :: CachingFetcher@Closed :: **rest**@Closed
– after update to field cache and post of $\mathsf{implSig}_{\mathsf{CachingFetcher}}$
Object@Default :: WebPageFetcher@Open :: CachingFetcher@Open :: **rest**@Closed

We now illustrate the situation at a virtual call site.

– Assume x has typestate
Object@Default :: WebPageFetcher@Closed :: CachingFetcher@Closed :: **rest**@Closed
– upcast for virtual call to Open
Object@Default :: WebPageFetcher@Closed :: **rest**@Closed
– virtual call to Open ($\mathsf{virtSig}_{\mathsf{WebPageFetcher}}$)
Object@Default :: WebPageFetcher@Open :: **rest**@Open
– after safe down-cast to CachingFetcher
Object@Default :: WebPageFetcher@Open :: CachingFetcher@Open :: **rest**@Open

The virtual call to Open changes all frames to typestate Open, since, dynamically, every frame of the object is changed.

5 Alias confinement

Any sound static checker of object invariants must be aware of all the references to an object in order not to miss any of the object's state transitions. We use a version of the adoption and focus model [4] for dealing with aliasing. We distinguish two modes for each object and statically track this mode for all pointers to objects. An object is either NotAliased, meaning that we statically know perfect aliasing information for this object (all must-aliases and no may-aliases). Otherwise, the object is MayBeAliased, in which case there can be arbitrary may-aliasing to the object among pointers tracked as MayBeAliased.

At allocation, an object is not-aliased. Objects can undergo arbitrary state changes when not-aliased, since we can statically track their state. Explicit deallocation of NotAliased objects is safe, but in this paper we don't discuss it further.

A NotAliased parameter guarantees to a method that it can access the object only through the given parameter or copies of the pointer that it makes, but the method cannot reach this object through any other access paths. At the same time, NotAliased guarantees to the caller that, upon return, the method will not have produced more aliases to the object. On fields, NotAliased specifies that the object with that field holds the only pointer to the referenced object. NotAliased objects can be transferred in and out of fields at any point. NotAliased objects

can be returned from methods, guaranteeing to the caller that no other aliases are still alive.

A NotAliased object can also *leak*, *i.e.*, transition to the MayBeAliased mode.[1] When an object leaks, all references to the object are considered MayBeAliased and may be copied arbitrarily. References to a MayBeAliased object are typestate-invariant; the object's typestate can no longer change. Thus, the moment an object leaks, its typestate is essentially frozen to the current typestate. We make this simplifying assumption to make the system tractable. A *focus* operation [4] can be used for temporarily changing the typestate of maybe-aliased objects, but for simplicity we ignore *focus* in this paper.

Since we allow a not-aliased object to leak, we must choose the rules for accessing the leaked object's not-aliased fields. There are three reasonable options:

Recursively leak Treat NotAliased fields of MayBeAliased objects as MayBeAliased. This approach is simple, works in the presence of concurrency, and allows both reading and writing of the field, but does not preserve the not-aliased status of sub-structures.

Leave not-aliased Allow access only via an atomic field-variable swap operation. This approach retains not-aliased status of such objects and also works in the presence of concurrency.

Disallow access to such fields Require a focus scope on the containing object to access such fields [4]. This approach requires extra locking in the presence of concurrency.

The formalism in the next section uses the first option.

6 Formal language

To formalize our approach, we present a small imperative, object-oriented language with a static typestate system. This language and type system form the kernel of a tool, called Fugue [6], which is a typestate checker for programming languages that compile to the .NET Intermediate Language, like C♯, Visual Basic, and Managed C++.

Besides the typestate aspects, the language is a standard object-oriented language, with classes, fields and methods. Each class has a single base class, unless it is the predefined class Object. The subclass relation ($\ll$) is the reflexive, transitive closure of the baseclass relation.[2] A class consists of virtual method declarations (`new`), method implementations (`impl`), fields, and typestates. To simplify the presentation, we assume all methods are virtual, sliding, and have distinct names.

The syntax of the formal language makes the checker's assumptions about the heap and values fully explicit in the form of pre- and postconditions at

[1] Leak corresponds to adoption in [4]. Here, we do not distinguish multiple adopters, but simply assume a single implicit adopter, namely the garbage collector.

[2] We assume the baseclass relation is non-cyclic, but the static semantics does not enforce it.

(program)	P	$::= class_1 \,..\, class_n \,\texttt{in}\, b$
(class)	$class$	$::= \texttt{class}\, C : D\ \{\overline{d}\}$
(declaration)	d	$::= \texttt{virt}\, M : \psi \mid \texttt{impl}\, M\{\overline{b}\} \mid \texttt{field}\, f \mid \texttt{state}\, s : \tau$
(method sig)	ψ	$::= \forall[\Delta](\rho_1..\rho_n); \Theta; \varphi \to \exists\rho.(\Theta'; \varphi')$
(code block)	b	$::= \psi\, \ell = \lambda(x_1 \,..\, x_n).stmt$
(statement)	$stmt$	$::= \texttt{let}\, x = e\, \texttt{in}\, stmt$
		$\mid \texttt{set}\, x.f = y\, \texttt{in}\, stmt$
		$\mid \texttt{pack}[C@s]\ x\, \texttt{in}\, stmt$
		$\mid \texttt{unpack}[C]\ x\, \texttt{in}\, stmt$
		$\mid \texttt{leak}\ x\, \texttt{in}\, stmt$
		$\mid \texttt{goto}\ tt$
(expression)	e	$::= x \mid c \mid y.f \mid \texttt{new}\, C \mid y.[C::]M(y_1..y_n)$
(targets)	tt	$::= \bullet \mid \ell[\rho_1..\rho_m](y_1..y_n)\ \texttt{when}\, A,\ tt \mid \texttt{return}\, x\ \texttt{when}\, A,\ tt$
(condition)	A	$::= \texttt{true} \mid x = c \mid x = y \mid x \neq c \mid x \neq y \mid A \wedge A \mid A \vee A$
		$\mid \mathsf{hastype}(x, C)$
(constant)	c	$::= 0, 1, ...$
(binding)	δ	$::= x : \rho \mid \ell : \psi \mid C@s : \tau \mid M : (\psi, C) \mid C::M : \psi \mid f : C$
(type env)	Γ	$::= \bullet \mid \delta, \Gamma$
(name env)	Δ	$::= \bullet \mid \rho, \Delta$
(heap)	Θ	$::= \bullet \mid \rho \mapsto (a, \sigma_C), \Theta$
(aliasing)	a	$::= 1 \mid + \mid \bot$
(object typestate)	σ_C	$::= \chi_C :: \mathbf{rest}@s \mid \chi_C :: \bullet$
(frames)	χ_C	$::= \chi_{\mathsf{baseclass}(C)} :: F_C$
	χ_{Object}	$::= F_{\mathsf{Object}}$
(frame)	F_C	$::= C@s \mid C\{f_1 : \rho_1 \,..\, f_n : \rho_n\}@s$
(frame typestate)	τ	$::= \exists\{f_1 : \rho_1 \,..\, f_n : \rho_n\}.(\Theta; \varphi)$
(value formula)	φ	$::= \texttt{true} \mid \rho = c \mid \rho = \rho' \mid \rho \neq c \mid \rho \neq \rho' \mid \varphi \wedge \varphi \mid \varphi \vee \varphi$
		$\mid \mathsf{hastype}(\rho, C)$
(value name)	ρ	
(typestate name)	s	
(block label)	ℓ	
(variable name)	x, y	
(class name)	C	
(field name)	f	
(method name)	M	

Fig. 8. Syntax of the language

every basic block in the method body. This allows us to separate checking from inference. In order to be practical, a system like Fugue infers the intermediate states inside a method, but inference is beyond the scope of this paper.

Fig. 8 contains the syntax of the language. A program is a set of classes and a single code block. Each class consists of virtual method declarations, method implementations, fields, and typestate interpretations τ. A typestate interpretation τ is an existentially closed predicate over the fields of a class frame. A method is a named set of labeled code blocks, where execution begins at the first block. Each code block is a closed function. Signatures ψ of methods and code

blocks have the form

$$\forall[\Delta](\rho_1 .. \rho_n); \Theta; \varphi \rightarrow \exists\rho.(\Theta'; \varphi')$$

where $\Theta; \varphi$ are the constraints on the heap and values on entry to the method or block, ρ names the result, and $\Theta'; \varphi'$ are the constraints on the heap and values (including the result) on exit of the method. The signature of a method is the signature of the first block in its body. The receiver is always the first parameter of a method. As usual, type equivalence is syntactic modulo renaming of bound variables.

There are no local variables. Data is passed between blocks through explicit parameters (think registers). Each code block ends in a set of control transfers to other code blocks or in a return, where each transfer is guarded by a condition A. When control reaches the end of the block, control follows an edge (chosen nondeterministically) whose condition is true at that point. The $\mathsf{hastype}(x, C)$ condition is an explicit type test that succeeds if x's dynamic type is C or a subclass of C. In conjunction with the rules in Figure 14, it allows for dynamic downcasts as well as recovering the static type after an upcast.

There are two kinds of method calls: virtual calls $y.M(...)$; and non-virtual calls $y.C::M(...)$ directly to the method M implemented in class C. We model object construction as a `new` expression followed by a non-virtual call to a constructor method.

The expressions leak, pack and unpack are non-standard constructs. A leak expression changes the mode of an object from not-aliased to maybe-aliased. The pack and unpack operations are used on not-aliased objects to coerce between the abstract typestate view and the concrete field view of a class frame of a particular object. In order to access (read or write) a field of a not-aliased object, the frame containing the field must be unpacked. Thus pack and unpack operations are required such that all accesses to fields are performed on unpacked frames. Packed frames are typically required at method boundaries. Aliased objects are never packed or unpacked.

Our type system assigns each value a symbolic name ρ, a form of singleton type. These names are used for pointers and scalars alike. Heaps Θ are mappings from pointer names ρ to an aliasing assumption a and the object's typestate σ_C. Formulae φ provide pure value constraints on ρ's. The typestate σ_C also specifies the static class type C of ρ. Alias assumptions take the forms 1 for not-aliased, $+$ for maybe-aliased, and $\perp$ for alias-polymorphic parameters. A heap mapping $\rho \mapsto (a, \sigma_C)$ is interpreted as a conditional mapping, predicated on $\rho \neq \mathsf{null}$.

There are implicit well-formedness conditions on heaps Θ regarding duplicates. If Θ contains duplicate mappings $\rho \mapsto (a_1, \sigma_1)$ and $\rho \mapsto (a_2, \sigma_2)$, then $a_1 = a_2 = +$ and $\sigma_1 = \sigma_2$, otherwise we consider the heap predicate unsatisfiable.

6.1 Static semantics

The static semantics enforces type and typestate safety. Figure 13 shows the rules for programs, classes and class members. Type environment Γ contains

$$\frac{\varphi' = \varphi \wedge \rho = c}{\Delta;\Gamma;\Theta;\varphi \vdash c : \rho;\Delta,\rho;\Theta;\varphi'}\ \textbf{const} \qquad \frac{\begin{array}{l}\Theta(\rho) = (1, C\{f_1{:}\rho_1 \mathinner{..} f_n{:}\rho_n\}@s :: \sigma)\\ \varphi \Rightarrow \rho \neq \texttt{null}\end{array}}{\Delta;\Gamma,y{:}\rho;\Theta;\varphi \vdash y.f_j : \rho_j;\Delta;\Theta;\varphi}\ \textbf{read1}$$

$$\frac{\Gamma(y) = \rho}{\Delta;\Gamma;\Theta;\varphi \vdash y : \rho;\Delta;\Theta;\varphi}\ \textbf{var} \qquad \frac{\begin{array}{l}\Theta(\rho) = (a, C@s :: \sigma) \quad a = + \vee a = \bot\\ \Gamma(C@s) = \exists\{f_1{:}\rho_1 \mathinner{..} f_n{:}\rho_n\}.(\Theta_2;\varphi_2)\\ \Theta' = \Theta, \overline{\Theta_2|_{\rho_j}}^{a} \quad \varphi \wedge \varphi_2|_{\rho_j} \quad \varphi \Rightarrow \rho \neq \texttt{null}\end{array}}{\Delta;\Gamma,y{:}\rho;\Theta;\varphi \vdash y.f_j : \rho_j;\Delta,\rho_j;\Theta';\varphi'}\ \textbf{read}$$

$$\frac{\begin{array}{l}\Theta' = \Theta, \rho{\mapsto}(1, \sigma_C@\mathsf{Zeroed})\\ \varphi' = \varphi \wedge \rho \neq 0\end{array}}{\Delta;\Gamma;\Theta;\varphi \vdash \texttt{new}\, C : \rho;\Delta,\rho;\Theta';\varphi'}\ \textbf{new} \qquad \frac{\Delta;\Gamma;\Theta;\varphi \vdash tt : \exists\rho'.(\Theta';\varphi')}{\Delta;\Gamma;\Theta;\varphi \vdash \texttt{goto}\, tt : \exists\rho'.(\Theta';\varphi')}\ \textbf{goto}$$

$$\frac{\begin{array}{l}\Gamma(y_i) = \rho_i \qquad i = 0..n\\ \Gamma([C::]M) = \forall[\Delta](\rho_1..\rho_n);\Theta_0;\varphi_0 \to \exists\rho'.(\Theta_1;\varphi_1)\\ \Theta;\varphi \vdash \Theta_0,\Theta_2;\varphi_0\end{array}}{\Delta;\Gamma;\Theta;\varphi \vdash y_0.[C::]M(y_1..y_n) : \rho';\Delta,\rho';\Theta_1,\Theta_2;\varphi \wedge \varphi_1}\ \textbf{call}$$

$$\frac{\begin{array}{c}\Delta;\Gamma;\Theta;\varphi \vdash e : \rho;\Delta';\Theta';\varphi'\\ \Delta';\Gamma,x{:}\rho;\Theta';\varphi' \vdash \mathit{stmt} : \exists\rho'.(\Theta';\varphi')\end{array}}{\Delta;\Gamma;\Theta;\varphi \vdash \texttt{let}\, x = e\, \texttt{in}\, \mathit{stmt} : \exists\rho'.(\Theta';\varphi')}\ \textbf{let}$$

$$\frac{\begin{array}{l}\Theta\ \ = \rho_x{\mapsto}(1, C\{f_1{:}\rho_1 \mathinner{..} f_j{:}\rho_j \mathinner{..} f_n{:}\rho_n\}@s :: \sigma), \Theta_1 \qquad \varphi \Rightarrow \rho_x \neq \texttt{null}\\ \Theta_2 = \rho_x{\mapsto}(1, C\{f_1{:}\rho_1 \mathinner{..} f_j{:}\rho_y \mathinner{..} f_n{:}\rho_n\}@s :: \sigma), \Theta_1\\ \Delta;\Gamma,x{:}\rho_x,y{:}\rho_y;\Theta_2;\varphi \vdash e : \exists\rho'.(\Theta';\varphi')\end{array}}{\Delta;\Gamma,x{:}\rho_x,y{:}\rho_y;\Theta;\varphi \vdash \texttt{set}\, x.f_j = y\, \texttt{in}\, e : \exists\rho'.(\Theta';\varphi')}\ \textbf{set1}$$

$$\frac{\begin{array}{l}\Theta(\rho_x) = (a, C@s :: \sigma) \qquad a = + \vee a = \bot \qquad \Theta(\rho_y) \neq (\bot, \_)\\ \Gamma(C@s) = \exists\{f_1{:}\rho_1 \mathinner{..} f_n{:}\rho_n\}.(\Theta_2;\varphi_2) \qquad \varphi \Rightarrow \rho_x \neq \texttt{null}\\ \Theta, \overline{\Theta_2}^{a};\varphi \wedge \varphi_2 \vdash \Theta, \overline{\Theta_2}^{a}[\rho_y/\rho_j];\varphi_2[\rho_y/\rho_j]\\ \Delta;\Gamma,x{:}\rho_x,y{:}\rho_y;\Theta;\varphi \vdash e : \exists\rho'.(\Theta';\varphi')\end{array}}{\Delta;\Gamma,x{:}\rho_x,y{:}\rho_y;\Theta;\varphi \vdash \texttt{set}\, x.f_j = y\, \texttt{in}\, e : \exists\rho'.(\Theta';\varphi')}\ \textbf{set}$$

$$\frac{\begin{array}{l}\Theta\ \ = \rho{\mapsto}(1,\ \sigma_C), \Theta_1 \qquad \sigma_C \text{ packed}\\ \Theta_2 = \rho{\mapsto}(+, \sigma_C), \Theta_1\\ \Delta;\Gamma,x{:}\rho;\Theta_2;\varphi \vdash e : \exists\rho'.(\Theta';\varphi')\end{array}}{\Delta;\Gamma,x{:}\rho;\Theta;\varphi \vdash \texttt{leak}\ x\, \texttt{in}\, e : \exists\rho'.(\Theta';\varphi')}\ \textbf{leak}$$

$$\frac{\begin{array}{l}\Theta = \rho{\mapsto}(1, C\{f_1{:}\rho_1 \mathinner{..} f_n{:}\rho_n\}@s' :: \sigma), \Theta_1\\ \Gamma(C@s) = \exists\{f_1{:}\rho_1 \mathinner{..} f_n{:}\rho_n\}.(\Theta_0;\varphi_0)\\ \Theta_1;\varphi \vdash \Theta_2,\Theta_0;\varphi_0\\ \Delta;\Gamma,x{:}\rho;\rho{\mapsto}(1, C@s :: \sigma),\Theta_2;\varphi \vdash e : \exists\rho'.(\Theta';\varphi')\end{array}}{\Delta;\Gamma,x{:}\rho;\Theta;\varphi \vdash \texttt{pack}[C@s]\ x\, \texttt{in}\, e : \exists\rho'.(\Theta';\varphi')}\ \textbf{pack}$$

$$\frac{\begin{array}{l}\Theta = \rho{\mapsto}(1, C@s :: \sigma), \Theta_1\\ \Gamma(C@s) = \exists\{f_1{:}\rho_1 \mathinner{..} f_n{:}\rho_n\}.(\Theta_0;\varphi_0)\\ \Theta_2 = \rho{\mapsto}(1, C\{f_1{:}\rho_1 \mathinner{..} f_n{:}\rho_n\}@s :: \sigma), \Theta_1, \Theta_0\\ \Delta,\rho_1..\rho_n;\Gamma,x{:}\rho;\Theta_2;\varphi \wedge \varphi_0 \vdash e : \exists\rho'.(\Theta';\varphi')\end{array}}{\Delta;\Gamma,x{:}\rho;\Theta;\varphi \vdash \texttt{unpack}[C]\ x\, \texttt{in}\, e : \exists\rho'.(\Theta';\varphi')}\ \textbf{unpack}$$

Fig. 9. Static semantics of statements and expressions

method signatures (both virtual and particular implementations), as well as frame typestate interpretations, and fields. In methods, Γ also contains local variables. Rules [**virt**] and [**impl**] enforce the relationship between virtual and implementation signatures of sliding methods described in Section 4. To simplify the class rules, we force classes to implement all virtual methods that could be invoked on them. An implementation can of course just call the base class method. The auxiliary function fn denotes the free names (ρ) of Γ, ψ, or τ.

Figure 9 contains the rules for statements and expressions. Judgment $\Delta; \Gamma; \Theta; \varphi \vdash \mathit{stmt} : \exists\rho.(\Theta'; \varphi')$ states that *stmt* is well formed in environment $\Delta; \Gamma; \Theta; \varphi$ and produces result ρ, in heap Θ' and value constraints φ'.

The judgments use a few notational shortcuts. The syntax $\sigma_C@s$ denotes the uniform object typestate $\mathsf{Object}@s :: ... :: C@s :: \bullet$. The syntax $F_C :: \sigma$ is a convenient pattern match to extract frame F_C from an object typestate.

Operation $\Theta|_\rho$ restricts the memory predicate Θ to the domain ρ (or the empty heap if not present). Similarly, $\varphi|_{\rho_j}$ restricts the value constraint to a conjunction of predicates on ρ_j only. Operation $\overline{\Theta}^a$ changes the aliasing of all not-aliased locations in Θ to a. It is used when accessing aliased or alias-polymorphic objects in order to adjust the aliasing of not-aliased fields. Accessing a not-aliased field of an alias-polymorphic parameter yields itself an alias-polymorphic object, thereby preventing it from changing, leaking, or escaping.

The judgment $\Theta; \varphi \vdash \Theta'; \varphi'$ is implication of heaps and value constraints. Heaps must be equivalent up to duplication of aliased locations and implication of formulae and typestates. Fig. 14 contains the implication rules.

The side condition σ_C packed in rule [**leak**] states that all frames of σ_C must be packed before the object can transition to an aliased mode.

Our decision on how to deal with aliased objects is visible in rule [**read**], governing access to fields of aliased objects. We re-instantiate the typestate predicate for the frame containing the field, since we assume that between any instruction, the field could change (this is conservative even in the presence of thread shared objects). Similarly, updating a field of an aliased object (rule [**set**]) requires that we prove the typestate predicates after substituting the new value name for the old, thereby guaranteeing that the update retains all invariants of the affected object.

This treatment makes explicit that field correlations cannot be observed of aliased objects, unless we extend the system with read-only fields or explicit focus scopes in which the object fields are not changed by the environment [4].

6.2 Soundness

Although we have no formal proof, we believe the system to be sound. We leave a study of its meta-theory for future work.

7 Discussion

Having given a formal definition for the language and its type system, we now discuss how it catches common programming errors, limits of the approach, and some extensions.

7.1 Example and errors that can be caught

Fig. 10 shows the methods CachingFetcher.Open and CachingFetcher.GetPage in the formal language. For brevity, we abbreviate *CachingFetcher* as *CF* and *WebPageFetcher* as *WPF* and drop all occurrences of $\bullet$ at the end of lists. These two methods represent common cases: Open changes the receiver's typestate and therefore its field invariants; GetPage assumes the field invariants of typestate Open and leaves the receiver in the same typestate.

We illustrate two kinds of programming errors that are common in mutator methods such as Open. First, the programmer of CachingFetcher::Open may forget to call the overridden method in the superclass, thereby not changing the typestate of the superclasses. In this case, at the method return point, the heap Θ would contain the entry

$$\rho_{\text{this}} \mapsto (1, \text{Object@Default} :: \text{WPF@Closed} :: \text{CF@Open} :: \textbf{rest}@\text{Closed})$$

which does not match that `post` clause, since frame WPF is Closed rather than Open.

Second, the programmer may fail to establish the object properties associated with the post-typestate Open of frame CachingFetcher, which is

$$\text{CF@Open} \equiv \exists\{\text{cache}{:}\rho_1\}.(\ \rho_1 \mapsto (+, \text{Object@Default} :: \text{Hashtable@Default})\ ;\ \rho_1 \neq \texttt{null})$$

Assuming the programmer sets field cache to null rather than a newly allocated Hashtable, an error manifests when applying rule [**pack**] to expression pack[CF@Open] of method CF::Open with the following bindings

$\Delta \equiv \rho_{\text{this}}, \bullet$
$\Gamma \equiv \text{this} : \rho_{\text{this}}, \bullet$
$\Theta \equiv \rho_{\text{this}} \mapsto (1, \text{Object@Default} :: \text{WPF@Open} :: \text{CF}\{\text{cache} : \rho_1\}@\text{Closed} :: \textbf{rest}@\text{Closed}), \bullet$
$\varphi \equiv \rho_{\text{this}} \neq \texttt{null} \wedge \rho_1 = \texttt{null}$

The critical premise in the [**pack**] rule is the implication

$$\Theta_1; \varphi \vdash \Theta_2, \Theta_0; \varphi_0$$

Given the current heap Θ_1 (minus the object being packed) and current value facts φ, we need to satisfy the heap Θ_0 and value invariants φ_0 associated with the typestate to which we are packing. (Θ_2 represents the unused portion of the heap.) In our hypothetical example, we have

$\Theta_1 \equiv \bullet\ ;\ \varphi \equiv (\rho_{\text{this}} \neq \texttt{null} \wedge \rho_1 = \texttt{null}) \qquad \Theta_2 \equiv \bullet$
$\Theta_0 \equiv \rho_1 \mapsto (+, \text{Object@Default} :: \text{Hashtable@Default})\ ;\ \varphi_0 \equiv (\rho_1 \neq \texttt{null})$

```
CF::Open {
  start : λ[ρ_this](this : ρ_this)
    pre  ρ_this ↦(1, Object@Default :: WPF@Closed :: CF@Closed :: rest@Closed);
         ρ_this ≠ null
    post ∃ρ'.(ρ_this ↦(1, Object@Default :: WPF@Open :: CF@Open :: rest@Closed);
              true)
      let _ = WPF::Open(this) in
      let h = new Hashtable in
      let _ = Hashtable::ctor(h) in
      unpack[CF] this in
      set this.cache = h in
      pack[CF@Open] this in
      goto return null when true
}

CF.GetPage {
  start: λ[ρ_this, ρ_path](this : ρ_this, path : ρ_path)
    pre  ρ_this ↦(⊥, Object@Default :: WPF@Open :: CF@Open :: rest@Open),
         ρ_path ↦(+, Object@Default :: String@Default);
         ρ_this ≠ null ∧ ρ_path ≠ null
    post ∃ρ'.(ρ_this ↦(⊥, Object@Default :: WPF@Open :: CF@Open :: rest@Open),
              ρ' ↦(+, Object@Default :: String@Default); ρ' ≠ null)
      let c = this.cache in
      let page = Hashtable::GetItem(c, path) in
      goto   missing[ρ_this, ρ_path, ρ_cache](this,path,c) when page = null,
             return page when page ≠ null

  missing: λ[ρ_this, ρ_path, ρ_cache](this : ρ_this, path : ρ_path, c : ρ_cache)
    pre  ρ_this ↦(⊥, Object@Default :: WPF@Open :: CF@Open :: rest@Open),
         ρ_path ↦(+, Object@Default :: String@Default),
         ρ_cache ↦(+, Object@Default :: Hashtable@Default);
         ρ_this ≠ null ∧ ρ_path ≠ null ∧ ρ_cache ≠ null
    post ∃ρ'.(ρ_this ↦(⊥, Object@Default :: WPF@Open :: CF@Open :: rest@Open),
              ρ' ↦(+, Object@Default :: String@Default); ρ' ≠ null)
      let page = WPF.GetPage(this, path) in
      let _ = Hashtable::Add(c, path, page) in
      goto return page when true
}
```

Fig. 10. Two CachingFetcher methods. We abbreviate CachingFetcher as *CF* and WebPageFetcher as *WPF* and reformat block heads for improved readability.

which is not satisfiable, since ρ_1 is null.

The code for method GetPage is more complicated because it has two basic blocks. There are two parts of the mechanics of checking this method that are worth pointing out. First, the code's correctness relies on the field invariants of typestate Open. GetPage treats the this parameter as alias-polymorphic rather

than NotAliased (to make the method more widely callable). Hence, the method can assume the typestate's field invariants, but cannot `unpack` the object and thereby change the field state. (The second premise of [**unpack**] requires that the object to unpack have alias mode 1.) The first block reads the field cache and binds it to the name c. This expression is checked with [**read**], rather than [**read1**], since this is possibly aliased. The conclusion of [**read**] yields a heap and value facts that are supplemented with the heap and value invariants of the field we are reading (namely, $\Theta, \overline{\Theta_2|_{\rho_j}}^{a}; \varphi \wedge \varphi_2|_{\rho_j}$). Here, we have $\rho_j \equiv \rho_{\text{cache}}$, $\Theta_2 \equiv \rho_{\text{cache}} \mapsto (+, \text{Object@Default} :: \text{Hashtable@Default})$ and $\varphi_2 \equiv \rho_{\text{cache}} \neq \texttt{null}$. The fact that $\rho_{\text{cache}} \neq \texttt{null}$ is needed to show the correctness of the next expression, the call to Hashtable::GetItem, which requires that its first argument not be null. The same proof-obligation exists in the second block at the call to Hashtable::Add.

Second, proving the correctness of this method relies on refining the value facts on conditional branches. The object ρ_{page} is the result of the call to Hashtable::GetItem, whose postcondition does *not* ensure that $\rho_{\text{page}} \neq \texttt{null}$. Hence, the legality of the return in start relies on the branch condition given in the `when` clause (in this case, page $\neq$ `null`). The use of conditions is explicit in the third premise of rule [**return**], where $\varphi \wedge \varphi_A$ is used to show the postcondition, and φ_A is the formula corresponding to condition A.

Finally, after discussing how the typing rules can be used to prove the correctness of method implementations, we look at how they catch such client errors as calling methods in the wrong order. Consider the following code sequence in which the programmer has forgotten to call Open before calling GetPage:

```
let f = new CachingFetcher in
let _ = CachingFetcher::ctor(f) in
let p = GetPage(f, "http://...") in ...
```

The critical premise of [**call**] is the implication $\Theta; \varphi \vdash \Theta_0, \Theta_2; \varphi_0$, requiring that the current heap Θ and value facts φ imply the heap Θ_0 and value precondition φ_0 of the called method (Θ_2 is the part of the heap unused by the method). The relevant facts here are $\Theta(\rho_{\text{f}}) = \text{Object@Default::WPF@Closed::CF@Closed} :: \textbf{rest}\text{@Closed}$, but the method expects $\Theta_0(\rho_{\text{f}}) = \text{Object@Default::WPF@Open} :: \text{CF@Open} :: \textbf{rest}\text{@Open}$. All typestates below the Object frame are thus in the wrong state.

7.2 Expressive power

This section examines the limits on the constraints that can be placed on object graphs using the object typestates formulated so far.

One can constrain any part of the object graph to form a tree using the not-aliased pointer predicates. Any field can be constrained to any unary predicate in the predicate language, including typestate predicates. Field values within the same class frame can be constrained arbitrarily using relational predicates (e.g., $x.f = x.g$, where f and g are within the same class frame.

Besides the restricted form of sharing constraints, the limitation of the object typestates described so far is that relations between fields of different frames or

different objects in the graph cannot be expressed, (e.g., $x.f = x.g.h$). The reason for this is that the only way to constrain the contents of an object referenced through a field is to specify its typestate. One cannot directly refer to $x.g.h$ in any formula. Typestates therefore fully abstract what can be observed about an object. Our formalization makes this explicit by modeling frame typestates with existential bindings for all fields:

$$C@s : \exists\{f_1{:}\rho_1..f_n{:}\rho_n\}.(\Theta, \varphi)$$

This states that the contents of frame C with typestate s is the set of field values ρ_i (one per field f_i), constrained by heap Θ and formula φ. The existential binding restricts the context to know nothing about the field values beyond the constraints φ.

Existentially abstracting only the field values implies that all frames referred to in Θ are packed (because the entire formula must not contain free value names). This choice is not fundamental and our framework can easily be extended to accommodate unpacked frames by allowing arbitrary existential abstraction of the form

$$C@s : \exists[\Delta].(\{f_1{:}\rho_1..f_n{:}\rho_n\}; \Theta, \varphi)$$

In such a formulation, constraints between different objects such as $x.f = x.g.h$ can be expressed, as long as the frame containing field h in object $x.g$ is unpacked in the typestate containing this constraint. Constraints between frames of the same object however remain outside this framework.

The typestates described so far are suited only to finite-state abstractions. For instance, typestates can enforce that Pop be called on a Stack object only after a call to Push or a non-emptiness test, but cannot enforce that Push be called at least as many times as Pop. Parameterized typestates (or dependent typestates) could support such counting abstractions similarly to the way they are enforced in ESC/Java using an integer ghost field.

7.3 Client and implementation views of typestates

There is a freedom in our formalization that we probably do not want. In this formulation, any code that has a not-aliased reference to an object may unpack and repack the object and thereby potentially change its typestate. In particular, this allows client code to change an object's typestate by directly accessing its fields rather than by calling its methods. If the programmer has made the object's representation private, this problem cannot arise, since the field assignments would be illegal. However, to promote programming hygiene, it is preferable to restrict client code to packing to the same typestate they unpacked. Only methods declared in the class whose frame is being packed (or one of its subclasses) are allowed to pack to different typestates.

```
[ TypeStates("InBounds ", " OutOfBounds") ]
interface IEnumerator
{
  [ Pre("InBounds ") ]
  object Current { get; }

  [ Post("InBounds ", WhenReturnValue=true),
    Post("OutOfBounds", WhenReturnValue=false) ]
  bool MoveNext ();

  [ Post("OutOfBounds") ]
  void Reset ();
}
```

Fig. 11. Using correlated return values to specify the IEnumerator class.

7.4 Correlating typestate and return values

In the typestate system presented so far, a method can specify only a single post-typestate for every parameter. This limitation prevents us from describing protocols in which a method can change a parameter to one of several post-typestates, correlated to the value of a returned status code. Fig. 11 shows a typestate specification for the popular .NET interface IEnumerator. To use an IEnumerator object, a program repeatedly calls MoveNext until it returns false and can call Current only when the latest call to MoveNext returned true. To capture this protocol, we need to correlate the object's typestate to the return value from MoveNext, using an extended feature WhenReturnValue=*constant*.

To support this feature in our formalism, we need to introduce typestate variables and allow quantification and constraints to range over such variables.

8 Related Work

Our work draws from several lines of research. Our aliasing approach has been heavily influenced by the work on alias types [7, 8], and region type systems [9, 10], in particular, the use of linear permissions and dependent types in some of these systems to control access to memory and to allow strong updates. This formulation allows for a natural imperative programming style, without the drawback of singly threading values as in traditional linear type systems.

Alias-polymorphic functions are closely related to the idea of let! by Wadler [11] and Boyland's alias burying [12]. See [4] for a detailed discussion of let! in the presence of imperative updates.

Our formulation of typestates for objects is novel. Previous work on typestates [2, 3] does not provide an interpretation to typestates as predicates over objects, nor did it consider the complications of subclasses.

The Fugue project shares the general goal of the work on the extended static checker ESC [1] to provide automatic checking of specifications for OO programs [13]. However, the two approaches differ in the following ways. Fugue focuses on a simple, sound programming model and a natural way to express

object properties via typestates. ESC is based on FOL and general theorem proving and is generally more expressive than Fugue. However, the expressiveness comes at a price. ESC does not aim for sound checking, nor the efficiency expected from a type checker. Although ESC has similar aliasing restrictions as described in this paper (NotAliased fields are called *pivot fields* in ESC), ESC lacks the ability to freeze typestates (object properties) as provided by leak statements. Thus, properties of arbitrarily aliased objects are difficult to capture across program abstractions.

The interaction of typestates and subclasses generally follows the notion of behavioral subtypes of Liskov et.al. [14]. Our formalism however does not support history constraints. On the other hand, unlike in Liskov's approach, our pre- and post-conditions are abstract predicates that allow subclasses to rely on strong properties not anticipated by the author of a supertype.

The use of a **rest** state in our object typestates is at first glance similar to the use of row-polymorphism to encode class types kov et.al. [14]. Our formalism however does not support history constraints. On the other hand, unlike in Liskov's approach, our pre- and post-conditions are in Objective ML [15]. However, object typestates have a very different purpose in that the **rest** state restricts the typestates of all possible extensions, whereas row-polymorphism does not restrict the types of fields in any extensions. Furthermore, our type system is based on name-based subclassing, not structural, where row-polymorphism is most useful.

Role analysis [16] attempts to capture the referencing behavior of structures and is similar to a typestate system. However, it does not address issues of subtyping and inheritance. It is also not clear how practical such systems can be made, in particular in light of [17].

9 Conclusion

We have attempted to strike a balance between expressiveness and practicality for specifying and checking object properties. Our approach supports the extension mechanism of class based programs in that it both allows subclasses to refine the interpretation of object typestates defined in superclasses, as well as to introduce entirely new typestates.

Acknowledgments

We thank the anonymous reviewers for their helpful comments and suggestions.

References

1. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J., Stata, R.: Extended static checking for Java. In: [18]
2. Strom, R.E., Yemini, S.: Typestate: A programming language concept for enhancing software reliability. IEEE TSE **12** (1986) 157–171

3. DeLine, R., Fähndrich, M.: Enforcing high-level protocols in low-level software. In: Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation. (2001) 59–69
4. Fähndrich, M., DeLine, R.: Adoption and focus: Practical linear types for imperative programming. In: [18] 13–24
5. Guttag, J.V., Horning, J.J.: Larch: Languages and Tools for Formal Specification. Springer-Verlag (1993)
6. DeLine, R., Fähndrich, M.: The fugue protocol checker: Is your software Baroque? Technical Report MSR-TR-2004-07, Microsoft Research (2004) URL: http://research.microsoft.com/~maf/fugue.
7. Smith, F., Walker, D., Morrisett, J.G.: Alias types. In: European Symposium on Programming. (2000) 366–381
8. Walker, D., Morrisett, G.: Alias types for recursive data structures. In: Proceedings of the 4th Workshop on Types in Compilation. (2000)
9. Tofte, M., Talpin, J.P.: Implementation of the typed call-by-value λ-calculus using a stack of regions. In: Conference Record of the 21st Annual ACM SSymposium on Principles of Programming Languages. (1994) 188–201
10. Crary, K., Walker, D., Morrisett, G.: Typed memory management in a calculus of capabilities. In: Conference Record of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press (1999)
11. Wadler, P.: Linear types can change the world! In Broy, M., Jones, C., eds.: Programming Concepts and Methods. (1990) IFIP TC 2 Working Conference.
12. Boyland, J.: Alias burying: Unique variables without destructive reads. Software—Practice and Experience **31** (2001) 533–553
13. Leino, K.R.M., Stata, R.: Checking object invariants. Technical Report #1997-007, DEC SRC, Palo Alto, USA (1997)
14. Liskov, B.H., Wing, J.M.: A behavioral notion of subtyping. ACM Transactions on Programming Languages and Systems **16** (1994) 1811–1841
15. Rémy, D., Vouillon, J.: Objective ML: an effective object-oriented extension to ML. Theory and Practice of Object Systems **4** (1998) 27–50
16. Kuncak, V., Lam, P., Rinard, M.: Role analysis. In: Conference Record of the 29th Annual ACM Symposium on Principles of Programming Languages. (2002)
17. Kuncak, V., Rinard, M.: Existential heap abstraction entailment is undecidable. In: Proceedings of the 10th International Static Analysis Symposium. (2003)
18. Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation. (2002)

Appendix

$\boxed{\Gamma;\Theta \vdash A : \varphi}$

$$\frac{\Gamma(x) = \rho \qquad \Theta(\rho) = (a, \sigma)}{\Gamma;\Theta \vdash \mathsf{hastype}(x, C) : \mathsf{hastype}(\rho, C)} \qquad \frac{A \in \mathtt{true}, =, \neq}{\Gamma;\Theta \vdash A : \Gamma(A)}$$

$$\frac{\Gamma;\Theta \vdash A_1 : \varphi_1 \qquad \Gamma;\Theta \vdash A_2 : \varphi_2}{\Gamma;\Theta \vdash A_1 \vee A_2 : \varphi_1 \vee \varphi_2} \qquad \frac{\Gamma;\Theta \vdash A_1 : \varphi_1 \qquad \Gamma;\Theta \vdash A_2 : \varphi_2}{\Gamma;\Theta \vdash A_1 \wedge A_2 : \varphi_1 \wedge \varphi_2}$$

Fig. 12. Static rules for conditions

$$\frac{\begin{array}{c}P = class_1..class_n \,\mathtt{in}\, b \\ \Gamma \vdash class_i \qquad i = 1..n \\ \Gamma \vdash b \\ \mathsf{fn}(\Gamma) = \emptyset\end{array}}{\Gamma \vdash P}\ \textbf{program} \qquad \frac{\begin{array}{c}\Gamma(M) = (\psi, C) \\ \Gamma(C{::}M) = \psi' \\ \mathsf{virtSig}_C(\psi') = \psi\end{array}}{\Gamma, C \vdash \mathtt{virt}\, M : \psi}\ \textbf{virt}$$

$$\frac{\begin{array}{c}\forall M.\Gamma(M) = (\psi, B) \wedge C \ll B \implies \mathtt{impl}\, M\{\overline{b}\} \in \overline{d} \\ \Gamma, C \vdash \overline{d}\end{array}}{\Gamma \vdash \mathtt{class}\, C : D\{\overline{d}\}}\ \textbf{class}$$

$$\frac{\begin{array}{c}\Gamma(C{::}M) = \psi_1 \qquad \Gamma(M) = (\_, B) \qquad \Gamma(B.M) = \psi' \qquad \psi_1 = \mathsf{implSig}_C(\psi') \\ b_i = \psi_i\, \ell_i \ldots \qquad i = 1..n \\ \mathsf{fn}(\psi_i) = \emptyset \qquad i = 1..n \\ \Gamma' = \Gamma, \ell_1 : \psi_1, .., \ell_n : \psi_n \\ \Gamma' \vdash b_i \qquad i = 1..n\end{array}}{\Gamma, C \vdash \mathtt{impl}\, M\{b_1..b_n\}}\ \textbf{impl}$$

$$\frac{\Gamma(f) = C}{\Gamma, C \vdash \mathtt{field}\, f}\ \textbf{field} \qquad \frac{\begin{array}{c}\Gamma(C@s) = \tau \qquad \mathsf{fn}(\tau) = \emptyset \qquad \Theta \not\ni \bot \\ \tau = \exists\{f_1{:}\,\rho_1\,..\,f_n{:}\,\rho_n\}.(\Theta; \varphi) \\ \Gamma(f_i) = C \qquad i = 1..n\end{array}}{\Gamma, C \vdash \mathtt{state}\, s : \tau}\ \textbf{state}$$

$$\frac{\begin{array}{c}\psi = \forall[\Delta](\rho_1..\rho_n); \Theta; \varphi \to \exists\rho.(\Theta'; \varphi') \\ \Delta; \Gamma, x_1{:}\,\rho_1..x_n{:}\,\rho_n; \Theta; \varphi \vdash e : \exists\rho.(\Theta'; \varphi')\end{array}}{\Gamma \vdash \psi\, \ell = \lambda(x_1..x_n).e}\ \textbf{block}$$

$\boxed{\Delta; \Gamma; \Theta; \varphi \vdash tt : \exists\rho'.(\Theta'; \varphi')}$

$$\frac{\begin{array}{l}\Gamma(\ell) = \forall[\overline{\rho}'](\rho_1..\rho_m); \Theta_0; \varphi_0 \to \exists\rho_r.(\Theta_1; \varphi_1) \qquad \Gamma; \Theta \vdash A : \varphi_A \\ \Gamma(y_i) = \rho_i[\overline{\rho}/\overline{\rho}'] \quad i = 1..m \\ \Theta; \varphi \wedge \varphi_A \vdash \Theta_0[\overline{\rho}/\overline{\rho}']; \varphi_0[\overline{\rho}/\overline{\rho}'] \\ \Theta_1[\overline{\rho}/\overline{\rho}']; \varphi_1[\overline{\rho}/\overline{\rho}'] \vdash \Theta'; \varphi' \\ \Delta; \Gamma; \Theta; \varphi \vdash tt : \exists\rho_r.(\Theta'; \varphi')\end{array}}{\Delta; \Gamma; \Theta; \varphi \vdash \ell[\overline{\rho}](y_1..y_m)\ \mathtt{when}\ A,\ tt : \exists\rho_r.(\Theta'; \varphi')}\ \textbf{label}$$

$$\frac{\begin{array}{l}\Gamma(x) = \rho' \qquad \Gamma; \Theta \vdash A : \varphi_A \qquad \Theta; \varphi \wedge \varphi_A \vdash \Theta'; \varphi' \\ \Delta; \Gamma; \Theta; \varphi \vdash tt : \exists\rho'.(\Theta'; \varphi')\end{array}}{\Delta; \Gamma; \Theta; \varphi \vdash \mathtt{return}\, x\ \mathtt{when}\ A,\ tt : \exists\rho'.(\Theta'; \varphi')}\ \textbf{return}$$

$$\frac{}{\Delta; \Gamma; \Theta; \varphi \vdash \bullet : \exists\rho'.(\Theta'; \varphi')}$$

Fig. 13. Well-formedness of programs, classes, methods, blocks, and goto targets

$\boxed{\Theta;\varphi \vdash \Theta';\varphi'}$

$$\frac{\varphi;\sigma \vdash \varphi';\sigma'}{\rho\mapsto(a,\sigma),\Theta;\varphi \vdash \rho\mapsto(a,\sigma'),\Theta;\varphi'} \qquad \frac{}{\bullet;\varphi \vdash \bullet;\varphi}$$

$$\frac{\varphi \Rightarrow \rho_1 = \rho_2}{\rho_1\mapsto(+,\sigma),\rho_2\mapsto(+,\sigma),\Theta;\varphi \vdash \rho_1\mapsto(+,\sigma),\Theta;\varphi} \qquad \frac{\varphi \Rightarrow \rho = \texttt{null}}{\rho\mapsto(a,\sigma),\Theta;\varphi \vdash \Theta;\varphi}$$

$$\frac{}{\rho\mapsto(+,\sigma),\Theta;\varphi \vdash \rho\mapsto(+,\sigma),\rho\mapsto(+,\sigma),\Theta;\varphi} \qquad \frac{\varphi \Rightarrow \rho = \texttt{null}}{\Theta;\varphi \vdash \rho\mapsto(a,\sigma),\Theta;\varphi}$$

$$\frac{}{\rho\mapsto(+,\sigma),\Theta;\varphi \vdash \Theta;\varphi} \qquad \frac{\Theta;\varphi \vdash \Theta';\varphi' \qquad \varphi' \Rightarrow \varphi''}{\Theta;\varphi \vdash \Theta';\varphi''}$$

$\boxed{\varphi;\rho;\sigma \vdash \varphi';\sigma'}$

$$\frac{\begin{array}{l}\varphi' = \varphi \wedge \mathsf{hastype}(\rho, C)\\ \varphi';\rho;\chi_B :: \mathbf{rest}@s \vdash \varphi'';\sigma_D\end{array}}{\varphi;\rho;\chi_B :: C@s :: \mathbf{rest}@s \vdash \varphi'';\sigma_D}\ \mathbf{upcast} \qquad \frac{\varphi;\chi_C \vdash \chi'_C}{\varphi;\rho;\chi_C :: \bullet \vdash \varphi;\ \chi'_C :: \mathbf{rest}@s}$$

$$\frac{\begin{array}{l}\varphi;\rho;\sigma_D \vdash \varphi';\chi_B :: \mathbf{rest}@s\\ \varphi' \Rightarrow \mathsf{hastype}(\rho, C)\\ \mathsf{baseclass}(C) = B\end{array}}{\varphi;\rho;\sigma_D \vdash \varphi';\ \chi_B :: C@s :: \mathbf{rest}@s}\ \mathbf{downcast} \qquad \frac{\varphi;\chi_C \vdash \chi'_C}{\varphi;\rho;\chi_C :: r \vdash \varphi;\ \chi'_C :: r}$$

$\boxed{\varphi;\chi_C \vdash \chi'_C}$

$$\frac{\varphi;\chi_B \vdash \chi'_B}{\varphi;\chi_B :: C@s \vdash \chi'_B :: C@s} \qquad \frac{}{\varphi;\bullet \vdash \bullet}$$

$$\frac{\varphi;\chi_B \vdash \chi'_B \qquad \varphi \Rightarrow \rho_i = \rho'_i}{\varphi;\chi_B :: C\{f_1{:}\rho_1..f_n{:}\rho_n\}@s \vdash \chi'_B :: C\{f_1{:}\rho'_1..f_n{:}\rho'_n\}@s}$$

Fig. 14. Implication rules

$$\mathsf{implSig}_C(\forall[\Delta](\rho_1..\rho_n);\Theta;\varphi \to \exists\rho.(\Theta';\varphi')) = \forall[\Delta](\rho_1..\rho_n);\mathsf{implSig}_C(\rho_1,\Theta);\varphi \to \exists\rho.(\mathsf{implSig}_C(\rho_1,\Theta');\varphi')$$

$$\mathsf{virtSig}_C(\forall[\Delta](\rho_1..\rho_n);\Theta;\varphi \to \exists\rho.(\Theta';\varphi')) = \forall[\Delta](\rho_1..\rho_n);\mathsf{virtSig}_C(\rho_1,\Theta);\varphi \to \exists\rho.(\mathsf{virtSig}_C(\rho_1,\Theta');\varphi')$$

$$\mathsf{implSig}_C(\rho,(\rho\mapsto(a,\sigma),\Theta)) = \rho\mapsto(a,\mathsf{implSig}_C(\sigma)),\Theta$$

$$\mathsf{virtSig}_C(\rho,(\rho\mapsto(a,\sigma),\Theta)) = \rho\mapsto(a,\mathsf{virtSig}_C(\sigma)),\Theta$$

$$\mathsf{implSig}_C(\chi_B :: C@s :: \mathbf{rest}@r) = \chi_B :: C@s :: \mathbf{rest}@r$$
$$\mathsf{implSig}_D(\chi_B :: C@s :: \mathbf{rest}@r) = \chi_B :: C@s :: \cdots :: D@s :: \mathbf{rest}@r \qquad D \ll C$$
$$\mathsf{virtSig}_C(\chi_B :: C@s :: \mathbf{rest}@r) = \chi_B :: C@s :: \mathbf{rest}@s$$

Fig. 15. Definitions of implSig and virtSig for sliding methods