A mode system for read-only references in Java

Mats Skoglund and Tobias Wrigstad
{matte,tobias}@dsv.su.se

Department of Computer and Systems Sciences (DSV),
Stockholm University/Royal Institute of Technology

1 Introduction

The use of references is typical in object-oriented programming. They are used for *e.g.* constructing compound objects, *i.e.* objects that hold references to other objects, and to perform in-place updates. Object sharing with references introduces problems since every object holding a reference to another can use it to invoke methods that modify the referenced object's state. This makes it harder to control the origin of changes to a specific object and thus makes it harder to reason about programs [8].

Since all references always give full access to their referenced objects' protocol, it is generally not possible to share an object by reference without risking that its mutator methods are used to modify it. In C++, this problem can be partly addressed by the use of const objects, const pointers and const methods although with some limitations. This is not possible in Java since it lacks a similar construct. Instead, approaches such as the proxy-pattern or more Java-specific constructs such as interfaces can be used to some extent but with some limitations, see *e.g.* [7] and [8].

We propose a new Java construct for read-only references that protects the transitive state of its referenced object. It is an improvement over C++'s const in that it is transitive and furthermore it does not suffer from some of the limitations of *e.g.* protection through interfaces or the limitations of the read-only constructs reviewed in related work.

2 Motivation

The local state of an object is made up of the values of its member variables and can be modified by *e.g.* assigning to a member variable in the object. With compound objects it is sometimes desirable to reason about the *transitive state* as opposed to the local state. The transitive state is made up not only of the member variables of the object, but also of the member variables' member variables and their member variables' member variables and so on. The transitive state of a compound object can be modified by either changing the local state of the object or by changing the local state of an object contained in the compound object's transitive state. We will use the term state to denote the transitive state throughout this paper.

A problem with compound objects arises when references to the compound object's sub-objects are shared with others. If all changes to a compound object's transitive state must be made via the object's protocol *e.g.* in order to keep the compound object consistent, sharing sub-objects with others can lead to errors. Changes made to a sub-object directly via a reference and not via the compound object's protocol bypasses any controls in the compound object's methods. Thus, such changes might invalidate internal invariants of the compound object leading to errors.

2.1 An example of a problem situation with sharing via references

The event mechanism in Java can be used for propagating state change notifications from one source object to multiple listener objects. When using events according to the JavaBeans API specification [5], one or more objects are normally included in the event object to represent the

state change to be notified. This makes the event object a compound object with references to sub-objects. An object contained in an event object is possibly shared with other compound objects (such as the source object).

It is strongly recommended in the JavaBean API specification that public access to an event object's members is denied and that accessor methods are used for exposing an event object's transitive state. However, if an object is exported from the event object through an accessor method by reference, the transitive state of the event object may still be changed underfoot if *e.g.* a mutator is invoked on the exported object (for a more general discussion, see [7] and [9]).

In Java, objects are always exported in such an "uncontrolled way" since we cannot impose "access rights" on the exported references – existing mutator methods may always be invoked. An exporter has no control over the use of the exported references and a receiver has no knowledge of whether it violates the exporter's intentions if it modifies an object via an incoming reference.

2.2 Related work

Approaches addressing the problem of exporting objects via references that can be used only to read the state of an object but not to modify it have been proposed in *e.g.* [7,11,6,9,8]. Most of these approaches use different kinds of read-only constructs to enforce the property that a reference cannot be used to modify the transitive state of its referenced object. The proposed constructs use variable annotations to indicate their read-only properties in the program. The read-only variable annotations are often accompanied by read-only annotations on methods to indicate which methods are safe to invoke on read-only variables.

Boyland et.al. [1] points out that the proposed read-only constructs are similar in their definitions but differ in semantics since the proposed constructs were defined for different purposes. Common properties of read-only references are that they cannot be used to modify the object that they refer and that only read-only methods can be invoked on them.

A common property of read-only methods is that they should not modify its enclosing object. Some of the read-only mechanisms require this last property to be transitive on compound objects *i.e.* a read-only method should not modify its enclosing objects transitive state. The motivation for transitive protection is that if the mechanism only protects the local state of an object it is sometimes possible to obtain a reference to a sub-object of a compound object that is not protected and use that reference to modify the compound object's transitive state and thus perhaps violate some invariant [6]. This is also the case if the mechanism would only protect a finite number of levels of the transitive state. Then it would be possible to obtain protected references from member variables' member variables in a number of steps and perhaps finally obtain an unprotected reference that can be used for modification.

Restrictions on ordinary programming We believe that the proposed constructs are too limiting on ordinary programming in many respects. For example, in order to protect the transitive state of a compound object, different read-only method constructs named *functional methods* and *clean messages* are defined in [8] and [9] respectively. These constructions have, in our point of view, some unnecessary limitations on ordinary programming. For example, neither a functional method nor a clean message can be used to modify objects, not even objects that are not part of the state of the protected object. This does not permit neither functional methods nor clean messages to *e.g.* perform in-place updates, even if *e.g.* the types of the parameters exclude the possibility that any argument is an alias to the enclosing object's transitive state. In our point of view, this is an unnecessary restriction since the read-onlyness should only concern the object that encloses the method.

Furthermore, in [8] a functional method may not return writeable references to instances created within the method which makes their construct unsuitable for *e.g.* the factory method pattern as described in [3].

In our point of, a read-only method should only protect its enclosing object's transitive state when invoked on a read reference but not necessarily when invoked on a write reference. For

example, a getter method should be able to return a writable reference to a member variable if invoked on a write reference since the object holding the write reference is already allowed to modify sub-objects of the referenced object. This means that there is no need for *e.g.* defining two different getter methods depending of the type of the reference, one that returns a read-only reference to an object and another that returns a writeable reference to the same object.

In [7], every programmer-supplied type has an implicit *readonly* supertype. The type of every variable declared as readonly will be changed to the corresponding readonly supertype consisting only of those methods that have been annotated by the programmer not to be state changing. As pointed out in [8] the use of implicit supertypes limits the use of protected member variables and methods, and furthermore, it creates a dependency on inheritance and a type system.

Validation of annotations The read-only constructs presented in [9] requires that the annotation of methods is done by the programmer and assumes that it is done *correctly*, *i.e.* methods that are read-only should be annotated as such by the programmer. This means that a programmer can, deliberately or undeliberately, annotate a method as read-only even if it modifies its enclosing object's state, making it possible to modify an object via read references with the problems described above.

In [7], as in [9], the programmer decides which methods should be read-only by annotations on the methods. However, no technique is presented that ensures that the annotations are correct with respect to the code. This means that methods may exist that are clearly read-only but cannot be used as such since they lack the readonly qualifier. Also, it is not clear whether methods that are annotated as read-only are validated not to modify their enclosing objects' state. If not, a method that modifies its enclosing object's state can possibly be invoked via read-only references.

C++'s const C++'s const can be used to some extent to achieve protection of objects. However, const methods only protect the local state of the enclosing object. To make the protection offered by const transitive, we need to include all objects in the transitive state in the local state. This can be done by storing an object in a variable as opposed to storing a reference to it. Thus, transitive protection with const precludes object sharing since all sub-objects need to be included in the enclosing object. While it is possible to store a reference to an object included in another object in a field, the referenced object will not achieve the transitive protection from changes from the object storing the reference. Furthermore, objects may not be passed to another as arguments to a method and stored in member fields without copying, precluding any intended sharing. The protection offered by const can easily be circumvented since const can be cast away.

Java's interfaces Interfaces in Java can be used to export parts of an object's protocol hiding any mutators and thus protecting an object from changes via a reference declared to be of the interface type. There are, however, a number of problems with references. The programmer must decide which methods that are mutating, which might be far from easy due to *e.g.* dynamic binding. Moreover, there is nothing to verify that the method implementing the method declaration in the interface does not modify its enclosing object's state. Also, there is nothing to prevent a subclass from overriding a supposedly non-mutating method with a mutator. Interfaces may also be circumvented by casting given that the actual class (or any superclass to that class) of the object is known by the programmer. Interfaces cannot be used for hiding private or protected methods from **this**.

3 A read reference approach to safe object exporting

We propose a mode system that allows exportation of objects without the risk of modification via read references similar to [7,11,6,8], but without some of their restrictions on ordinary programming.

The mode system works by *mode annotations* on variables and methods which control the flow of references in the system in some respects. Each variable, formal parameter, method etc. is associated with a mode qualifier (a mode for short). The modes controls valid assignments, valid method invocations etc.

The mode annotations on variables control what kind of references may be held by the variable. We distinguish between two kinds of references, *read references* and *write references*. Simply put, a write reference is a regular Java reference and a read reference is a reference that is never used for modification of its referenced object (including retrieval of write references that may in turn be used for modification).

The system should be statically checkable[1]. This means that a program can be statically validated to behave correctly with respect to the modes of the references held by the variables and the modes of the methods. Without the **caseModeOf** construct (see below, page 5), there is no need for an actual run-time representation of modes.

A formalisation of the static mode-checking system is shown in Appendix A. We show the rules for well-modedness in a modification of ClassicJava [2]. We do not show any operational semantics since these should be pretty straight-forward.

3.1 The core annotations

The core annotations are read/write annotations which are used to annotate member variables, local variables, formal parameters, method returns and methods. Basically a method declared as read (a read method for short) may not modify its enclosing object's state. Methods declared as write more or less behave like ordinary Java methods. These loose definitions will be refined below.

A variable or formal parameter declared as read (read variables for short) may not be used to invoke write methods and may thus not be used for modification of its referenced object[2]. Note that read variables may be assigned to, in Java terms they are not `final`. Variables declared as write may be used to invoke both read and write methods. The mode of the method return controls the mode of the reference returned by the method. A write variable always holds write references and a read variable always holds read references.

3.2 Approximate annotations

In addition to the core annotations, we have any/context annotations. These annotations do not apply to methods and differ from the core annotations in that these are not really modes but approximations of modes in compile-time (*i.e.* the content of an any or context variable may be either a read reference or a write reference in run-time). A context annotation of a variable means that the mode of the variable is the same as the mode of the *context*. For the moment, let the mode of the context be the same as the mode of the method. Thus, a context variable will be treated as read in a read method and as write in a write method. The last property requires that only write variables be assigned to context variables. Otherwise a read reference stored in a context variable could be wrongfully treated as write reference in a write context.

The any annotation is similar to the context annotation but does not depend on the mode of the context in the same way. Any reference may be assigned to an any variable regardless of its mode. For member variables in a write context, all references in any variables have the same mode as when they were stored in the variable. For member variables in a read context, all any variables will hold read references. For non-member variables, the mode of a reference held by an any variable is always the same as when it was stored in the variable regardless of the mode of the context.

Statically, an any variable must be treated as a read variable since it may contain a read reference whose 'readness' would be broken were it possible to invoke write method on it. Dynamically,

[1] The **caseModeOf** (see below, page 5) construct enables static checkability by performing run-time checks still ensuring that the program is well-behaved with respect to modes.

[2] For simplicity, we disregard from public variables.

an any variable may be converted to a write variable (*i.e.* treated as if it held a write reference) if the mode of the reference stored in the variable is really write, similar to a regular downcast.

3.3 Declarations

Member variables must be declared as either read, context or any. They may not be declared as write since write variables always hold write references which could be used to change the referenced object in read methods[3]. The **this** variable is context.

Local variables and method returns must be declared as either read, write, context or any. Variables declared as context must always refer to members of the enclosing object, *i.e.* may only be assigned from members or other context variables.

Formal parameters must be declared as either read, write or any. They may not be declared as context since the mode of the argument passed to the methods may not have the same mode as the method's context.

Context We say that all statements in a method body executes in the same *context*. The mode of the context is the run-time mode of the reference used to invoke the method. For a write method, this is always write since only write references may be used to invoke write methods. For a read method, the context is either read or write since references of both modes may be used for invoking read methods. Thus, in a read method the mode of a context variable may be either read or write. In compile-time, we must assume that the mode of a context variable in a read method is read by conservativness. However, we supply a dynamic construct that can be used to determine the actual mode of the context in run-time, increasing the flexibility of read methods. This construct is called **caseModeOf** (see below).

Note that this means that the definition of a read method above has been slightly altered. The new definition of a read method states that the a read method does not modify its enclosing object's state when invoked via a read reference. When invoked via a write reference, a read method may behave as a write method since the mode of the references held by any variables and context variables may now be write. Thus, the annotations on methods state the possible modes of the context analogous with the approximate annotations on variables and parameters. For example, the **caseModeOf** construct may change a method's behaviour according to the mode of the context.

Adding optional dynamics The **caseModeOf** construct has two purposes. It can dynamically check the mode of an any variable to allow it to be used as a write variable if it holds a write reference, and it can dynamically check the mode of the context to allow context variables to be used as write variables if the mode of the context is write.

The layout of the **caseModeOf** construct is similar to Java's switch-statement. It consists of a check on a variable and two blocks, the read block and the write block. The checked variable will be assumed to be write in the write block and read in the read block. It will not proceed to any other block. In run-time, the mode of the reference held by the variable will be checked and the corresponding block executed.

The **caseModeOf** construct is optional in the sense that it can be removed along with the any qualifier. This yields a system which is statically checkable without any need for dynamic checks or run-time representation of modes but with the drawback that it becomes less flexible.

Methods A method must be declared as either read or write. For simplicity, we require that all method overriding preserves the mode of all formal parameters and also the mode of the method.

A read method treats all member variables as `final` and statically assumes that all context variables contain read references, *i.e.* must be treated as read variables. This ensures that

[3] Other approaches are of course possible, such as to prevent names of write members to appear in bodies of write methods.

the enclosing object's state cannot be changed by the read method since any accessible member variable is read. Inside the write block of a **caseModeOf** statement on a member variable in a read method, the member may be modified since this block will only be executed in a write context, *i.e.* when the method was invoked via a write reference and it thus may modify its enclosing object's state.

A write method treats all context member variables as write and may thus change or export write references to the state of its enclosing object. Variables declared as read or any must still be treated as read. An any variable may, however, be modified in the write block of a **caseModeOf** statement testing the mode of its contained reference.

Parameters to methods and aliasing Note that a read method may modify the state of its enclosing object even in a read context if a write reference to the state is passed in as an argument. We allow this since the owner of that reference may use it for modification outside the method meaning that any protection from this situation inside the method does not significantly reduce the possibility of changes via the existing write reference. Moreover, it is arguable that the presence of a write reference outside the object should indicate that the changes to the state of the object should be allowed.

To avoid this, a possible solution is to require that all formal parameters to read methods are annotated with read (too restrictive in our sense). Another approach is to require formal parameters to be annotated with any and dynamically check all arguments for aliasing converting any incoming aliases to read references. These techniques could be somewhat optimised by *e.g.* checking if the possible types of possible arguments overlap with the possible state of this.

3.4 A brief example

```
1  public class SecNewThermometerEvent {
2    private read Object source;
3    private context Thermometer thermometer;
4    public void setSource(read Object s):write {
5      this.source = s;
6    }
7    public void setThermometer(write Thermometer th):write {
8      this.thermometer = th;
9    }
10   public read Object getSource():read {
11     return this.source;
12   }
13   public context Thermometer getThermometer():read {
14     return this.thermometer;
15   }
16 }
17 ...
18 write SecNewThermometerEvent event = new SecNewThermometerEvent();
19 event.setSource(read this);
20 event.setThemometer(thermometer);
21 radiators.notify(read event);
22 ...
```

Figure1. Example of event annotated with modes

The code above below, admittedly somewhat contrived for pedagogical reasons, uses the mode system in the construction and use of an event class. The event is used to notify radiator objects

in a room object that a thermometer object has been inserted in the room. The example is written in a Java language extended with the mode qualifiers on variables and methods.

A room object that holds a write reference to the newly inserted thermometer creates a new `SecNewThermometerEvent` object and assigns it to the write variable `event` (18). Then the room invokes `setSource` (19) that assigns the `source` variable in `event` (5). It then invokes the `setThermometer` method passing the thermometer as a parameter (20) and a reference to the thermometer will be stored in the `thermometer` variable in `event` (8). Finally, the radiators in the room are notified about the newly inserted thermometer by invocation of `notify` (21), passing `event` as read so it will be protected.

It is possible to assign `th` to `thermometer` (8) even though the parameter `th` is a write variable and the member variable `thermometer` is declared context. This is since a variable declared as context can be treated as a write variable in a write method and since the `setThermometer` method is declared as write (7) the assignment is allowed. Since the event object is sent to its receivers only as read (21) and since a context variable is always treated as read in a read context, the receivers will only be able to obtain read references to the thermometer object.
The member variable `source` holds a reference to the creator of the event object so that the receivers should be able to *e.g.* determine the origin of the event. This can be done by for example comparing the identity of the object held by `source` with another object's identity. Since `source` is declared as read (2) it is safe to export it from the event via the `getSource()` method rest assured that no receiver will be able to modify it via the exported reference. The formal parameter to `setSource`, `s`, is declared as read (4) since it will be stored in the read variable `source` (5) that in this example should never be allowed to be used to modify the creator.

The `thermometer` variable holds a reference to the thermometer inserted into the room. The `thermometer` is declared context (3), which means that since the event is sent to its receivers only via read references, the receivers may not obtain write references to the thermometer from the event. Neither may the receivers invoke the setters on the event since they are both write methods (4)(7).

4 Conclusions

We have presented a system for controlling references in Java. We distinguish between read references and write references. Our formal mode system is statically checkable and ensures that references exported as read will never be used to modify its referenced object. For increased flexibility, we also introduced an optional dynamic check. We also distinguish between read methods and write methods where read a method never modifies the state of its enclosing object when invoked on a read references. The statically checkable rules ensure that all methods in a program are annotated correctly.

The system is designed for class-based object-oriented programming languages and should be realisable not only in Java. For Java however, it seems reasonable to implement the mode system as an extension of the type system since all variables, method return declarations and formal parameters are associated with modes and since modes also further define the range of values a variable can assume. The presence of inheritance and method overriding in Java affects the criteria of the write method. Here, for simplicity, we require method overriding to preserve the modes of the overridden method. Thus, in addition to the original criteria, a method is a write method also if it overrides a write method.

The mode system is formalised as a type system with judgments, mode rules etc. Our work in progress is the completion of the system with operational semantics for the dynamic behaviour of *e.g.* **caseModeOf** and completion of the proofs of the mode system.

We also plan to extend our Java subset to get closer to the Java Language Specification [4].

References

1. J. Boyland, J. Noble, and W. Retert. Capabilities for aliasing: A generalization of uniqueness and read-only. Accepted to ECOOP 2001. Under revision.

2. M. Flatt, S. Krishnamurthi, and M. Felleisen. A programmer's reduction semantics for classes and mixins. Technical Report TR 97-293, Rice University, 1997, revised 6/99. Original in Formal Syntax and Semantics of Java, LNCS volume 1523 (1999), Available from `http://www.cs.rice.edu/CS/PLT/Publications/tr97-293.pdf`.
3. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software.* Addison-Wesley Publishing Company, 1995.
4. J. Gosling, B. Joy, G. L. Steele Jr., and G. Bracha. *The Java Language Specification Second Edition.* Addison-Wesley Publishing Company, 2000. Available from `http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html`.
5. G. Hamilton, editor. *JavaBeans.* Sun Microsystems Inc, 1997. Available from `http://java.sun.com/products/javabeans/docs/spec.html`.
6. J. Hogg. Islands: Aliasing protection in object-oriented languages. In A. Paepcke, editor, *OOPSLA '91 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications*, pages 271–285. ACM Press, 1991.
7. G. Kniesel and D. Theisen. Java with transitive access control. In *IWAOOS'99 – Intercontinental Workshop on Aliasing in Object-Oriented Systems. In association with the 13th European Conference on Object-Oriented Programming (ECOOP'99)*, June 1999. Available from `http://cui.unige.ch/~ecoopws/iwaoos/papers/index.html`.
8. P. Müller and A. Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen, 2001. Available from `http://www.informatik.fernuni-hagen.de/pi5/publications.html`.
9. J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In E. Jul, editor, *ECOOP '98—Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 158–185. Springer, 1998.
10. M. Skoglund and T. Wrigstad. A mode system for read references. Technical report, Department of Computer and Systems Sciences, Stockholm University/Royal Institute of Technology, 2001. To appear.
11. B. Stoustrup. *The C++ Programming Language Third Edition.* Addison-Wesley Publishing Company, 1997.

A Mode-checking rules

The rules are shown in Appendixes A.2-A.4 and briefly explained below. The system is a modification of ClassicJava, [2]. Included here are the most integral parts, a complete description of the mode-checking system is included in [10]. The most significant changes with respect to ClassicJava are the integration of the mode system, the **caseModeOf**-statement and the treatment of local variables.

For simplicity, we assume the following: All classes are declared only once, no inheritance chains are cyclic, no method is declared in a class more than once, no field is declared in a class more than once, every class is an extension of another declared class or Object, parameter names and local variable names in the same method do not conflict, method and field overriding preserves all modes and types. Note that we also assume that all methods bodies end with an expression. The result of a method invocation is the result of the last expression in the method body.

We write $P, F, m_c \vdash \langle p \rangle$ to denote that given the program text P, a mapping from local variable names to modes and types F and the mode of the context m_c, we may conclude $\langle p \rangle$. We write $e : (m, t)$ to denote that the expression e can be associated with a mode m and a type t. Note that in rule amcmo, we add an additional premise on the LHS of the $\vdash$ stating the existence of a field *fd* with mode m and type t in the type *t'* replacing an existing field in *t'* with the same name.

All expressions return values. The expressions $vn = e$ and **this**.*fd* $= e$ both return the value of the RHS allowing the expressions to be chained and used as arguments to methods.

We subscript $\vdash$ as $\vdash_{\mathsf{d}}$ for validation of class declarations, as $\vdash_{\mathsf{m}}$ for validation of method declarations, as $\vdash_{\mathsf{t}}$ for validation of mode/type pairs and as $\vdash_{\mathsf{s}}$ for type subsumption and mode conversion. Plain $\vdash$ denotes checks of expressions, statements or sequences of statements and expressions.

In addition to the non-terminals of the abstract syntax, we use the m to denote any mode (*e.g.* $m \in \{\mathsf{read}, \mathsf{write}, \mathsf{any}, \mathsf{context}\}$). We use standard '′' (prime) notation and subscription to

distinguish between meta variables of the same kind, *e.g.* m and m' are both modes, not necessarily the same. The symbol m_c always denotes the mode of the current context, read or write.

A.1 Explanation of the integral parts of the integral rules

Expressions Rule null states that null can be given any mode and thus be stored in every variable or field. Rule new states that references to objects created by new have mode write. Rule rcast states that any result of an expression may be cast to read. Rule lget states that the result of an access of a local variable has the variable's declared mode and type; $var \in dom(F)$ states that var is declared in F and $F(var)$ retrieves the mode and type for a variable name. Rule lset states that an assignment to a local variable requires that both sides have the same mode or that the RHS is a member with mode write and the LHS has mode context. Rule fget states that the result of an access to a field in **this** has the field's declared mode if the mode of **this** is write, else it has the mode read. Rule fset states that assignments to members requires that the mode of the context is write and that the mode on the RHS can be converted to the declared mode of the field on the LHS. Rule call states that write methods may only be invoked on write references, read methods may be invoked on a reference regardless of its mode. It also states that the modes of the arguments must be possible to convert to the same modes as the formal parameters.

Declarations Rule meth states that a method is well-moded if its body is well-moded with its declared mode as the mode of the context. The rule also states that if the method overrides a method with mode write, the method must be declared as write even if it is valid when the declared mode is read. Note that this does not allow methods that would compile as read methods to be declared as write methods and vice versa, except for the case of method overriding because of the difference between the most specific method selected in compile-time and the actual method bound to in run-time.

Statements Rule amcmo states that **caseModeOf** on a member variable declared as any is valid if the expressions in the write-block are valid with the variable as write and vice versa for the read-block. Rule alcmo states the same, but for local variables. Rule ccmo states that **caseModeOf** on a variable declared as context is valid if the expressions in the write block are valid with in with the mode of the context as write and vice versa for the read block.

Sequence Rule sseq states that a sequence of statements or expressions is well-moded if all substatements or subexpressions are well-moded. Note the s here, meaning that statements may be contained in sequences. Also note that an expression is also a statement.

Mode and type rules Rule mconv states that a mode m may be converted to m, any or, if m is context, to the mode of the current context, m_c. Rule mode/type states that mode and type pair is valid if the mode is in the set {read, write, any, context} and the type is declared in P. The rule sub states that the result of an expression with mode m and type t can be subsumed to be of a supertype to t and another mode to which m may be converted. Note that conv/sub accepts sequences of statements. While this might seem slightly unorthodox, it allows smooth treatment of method bodies, which always end with an expression.

A.2 Abstract syntax

$class$:= **class** t **extends** t { $field^*$ $meth^*$ }
$field$:= m_f t vn
$meth$:= m_l t $md((m_a\ t\ vn)^*)$:m_m { $body$ }
$local$:= m_l t vn
$body$:= $local^*$ s_{seq}
s_{seq} := s; | s; s_{seq}
e := null | **new** c | var | $vn = e$ | **this**.fd
| **this**.$fd = e$ | $e.md(e^*)$ | **read** e
s := e | **caseModeOf** ((**this**.fd | var)) {**w:** s_{seq} **r:** s_{seq}}

t := a classname or Object
m_f := read | any | context
m_l := read | write | any | context
m_a := read | write | any
m_m := read | write
vn := a variable name
var := vn | **this**
fd := a field descriptor
md := a method descriptor

A.3 Explanation of symbols (denotes relations)

$<:$ $t <: t'$ iff t' is a superclass to t or t'.
$\in_{\mathsf{c}}$ $t \in_{\mathsf{c}} P$ iff t is a class declared in program P.
$\in_{\mathsf{f}}$ $(m, t, fd) \in_{\mathsf{f}} t'$ iff field fd with mode m and type t is declared in class t'.
FA(e) FA(e) iff $e \equiv$ **this** or $e \equiv$ **this**.fd
$\in_{\mathsf{m}}$ $(m', t', md, ((m, t)\ldots), m'') \in_{\mathsf{m}} t'$ iff method md with return mode/type- pair (m', t') and formal parameters $vn\ldots$ with mode/type-pairs $(m, t)\ldots$ method mode m'' is declared in class t''.

A.4 Mode/type elaboration

$$\text{null}\ \frac{P \vdash_{\mathsf{t}} (m, t)}{P, F, m_c \vdash \mathsf{null} : (m, t)} \qquad \text{new}\ \frac{t \in_{\mathsf{c}} P}{P, F, m_c \vdash \textbf{new}\ t : (write, t)} \qquad \text{rcast}\ \frac{P, F, m_c \vdash e : (m, t)}{P, F, m_c \vdash \textbf{read}\ e : (\mathsf{read}, t)}$$

$$\text{lget}\ \frac{var \in dom(F)}{P, F, m_c \vdash var : F(var)} \qquad \text{lset}\ \frac{\begin{array}{c}P, F, m_c \vdash vn : (m', t) \quad P, F, m_c \vdash_{\mathsf{s}} e : (m, t) \\ m = m' \vee (\text{FA}(e) \wedge m = \mathsf{write} \wedge m' = \mathsf{context})\end{array}}{P, F, m_c \vdash vn = e : (m, t)}$$

$$\text{fget}\ \frac{\begin{array}{c}P, F, m_c \vdash_{\mathsf{s}} \textbf{this} : (m', t) \qquad (m'', t', fd) \in_{\mathsf{f}} t \\ m' = \mathsf{write} \Rightarrow m = m'' \qquad m' \neq \mathsf{write} \Rightarrow m = \mathsf{read}\end{array}}{P, F, m_c \vdash \textbf{this}.fd : (m, t)} \qquad \text{fset}\ \frac{\begin{array}{c}P, F, m_c \vdash_{\mathsf{s}} \textbf{this} : (\mathsf{write}, t) \\ (m', t', fd) \in_{\mathsf{f}} t \qquad P, F, m_c \vdash_{\mathsf{s}} e : (m', t')\end{array}}{P, F, m_c \vdash \textbf{this}.fd = e : (m', t')}$$

$$\text{call}\ \frac{\begin{array}{c}m' = \mathsf{write} \vee m'' = \mathsf{read} \qquad P, F, m_c \vdash_{\mathsf{s}} e : (m', t') \\ P, F, m_c \vdash_{\mathsf{s}} e_j : (m_j, t_j) \text{ for } j = [1, n] \\ (m, t, md, ((m_1, t_1)\ldots(m_n, t_n)), m'') \in_{\mathsf{m}} t'\end{array}}{P, F, m_c \vdash e.md(e_1\ldots e_n) : (m, t)} \qquad \text{mode/type}\ \frac{\begin{array}{c}t \in_{\mathsf{c}} P \\ m \in \{\mathsf{read}, \mathsf{write}, \mathsf{any}, \mathsf{context}\}\end{array}}{P \vdash_{\mathsf{t}} (m, t)}$$

$$\text{mconv}\ \frac{m = m' \vee m' = \mathsf{any} \vee (m = \mathsf{context} \wedge m' = m_c)}{m_c \vdash_{\mathsf{m}} m \leq_M m'} \qquad \text{class}\ \frac{\begin{array}{c}P \vdash_{\mathsf{t}} (m_j, t_j) \text{ for } j = [1, n] \\ P, c \vdash_{\mathsf{m}} meth_k \text{ for } k = [1, p]\end{array}}{\begin{array}{l}P \vdash_{\mathsf{d}} \textbf{class}\ c\ \ldots\ \{m_1\ t_1\ fd_1\ldots \\ \quad m_n\ t_n\ fd_n\ meth_1\ldots meth_p\}\end{array}}$$

$$\text{conv/sub}\ \frac{\begin{array}{c}P, F, m_c \vdash s_{\mathsf{seq}} : (m', t') \\ m_c \vdash_{\mathsf{m}} m' \leq_M m \qquad t' <: t\end{array}}{P, F, m_c \vdash_{\mathsf{s}} s_{\mathsf{seq}} : (m, t)} \qquad \text{meth}\ \frac{\begin{array}{c}t_0 <: t_1 \qquad P \vdash_{\mathsf{t}} (m, t) \qquad m'' \in \{\mathsf{read}, \mathsf{write}\} \\ P \vdash_{\mathsf{t}} (m_j, t_j) \text{ for } j = [1, n+p] \qquad P, \kappa, m'' \vdash_{\mathsf{s}} s_{\mathsf{seq}} : (m, t) \\ (m, t, md, ((m_1, t_1)\ldots(m_n, t_n)), \mathsf{write}) \notin_{\mathsf{m}} t_1 \Rightarrow m'' = m' \\ \kappa = [\textbf{this} : (\mathsf{context}, t_0), v_1 : (m_1, t_1)\ldots v_{n+p} : (m_{n+p}, t_{n+p})]\end{array}}{\begin{array}{c}P, t_0 \vdash_{\mathsf{m}} m\ t\ md(m_1\ t_1\ v_1\ldots m_n\ t_n\ v_n) : m' \\ \{m_{n+1}\ t_{n+1}\ v_{n+1}\ \ldots m_{n+p}\ t_{n+p}\ v_{n+p}\ s_{\mathsf{seq}}\}\end{array}}$$

$$\text{amcmo}\ \frac{\begin{array}{c}P, F, m_c \vdash \textbf{this}.fd : (\mathsf{any}, t) \qquad P, F, m_c \vdash \textbf{this} : (\_, t_1) \\ (\mathsf{write}, t, fd) \in_{\mathsf{f}} t_1, P, F, m_c \vdash s'_{\mathsf{seq}} : (m', t') \\ (\mathsf{read}, t, fd) \in_{\mathsf{f}} t_1, P, F, m_c \vdash s''_{\mathsf{seq}} : (m'', t'')\end{array}}{\begin{array}{c}P, F, m_c \vdash \textbf{caseModeOf}\ (\textbf{this}.fd) \\ \{\textbf{w:}\ s'_{\mathsf{seq}}\ \textbf{r:}\ s''_{\mathsf{seq}}\} : (\mathsf{any}, t)\end{array}} \qquad \text{alcmo}\ \frac{\begin{array}{c}P, F, m_c \vdash vn : (\mathsf{any}, t) \\ P, F[vn : (\mathsf{write}, t)], m_c \vdash s'_{\mathsf{seq}} : (m', t') \\ P, F[vn : (\mathsf{read}, t)], m_c \vdash s''_{\mathsf{seq}} : (m'', t'')\end{array}}{\begin{array}{c}P, F, m_c \vdash \textbf{caseModeOf}\ (vn) \\ \{\textbf{w:}\ s'_{\mathsf{seq}}\ \textbf{r:}\ s''_{\mathsf{seq}}\} : (\mathsf{any}, t)\end{array}}$$

$$\text{ccmo}\ \frac{\begin{array}{c}P, F, m_c \vdash e : (\mathsf{context}, t) \\ P, F, \mathsf{write} \vdash s'_{\mathsf{seq}} : (m', t') \qquad P, F, \mathsf{read} \vdash s''_{\mathsf{seq}} : (m'', t'')\end{array}}{P, F, m_c \vdash \textbf{caseModeOf}\ (e)\ \{\textbf{w:}\ s'_{\mathsf{seq}}\ \textbf{r:}\ s''_{\mathsf{seq}}\} : (\mathsf{context}, t)} \qquad \text{sseq}\ \frac{\begin{array}{c}P, F, m_c \vdash s : (m, t) \\ P, F, m_c \vdash s_{\mathsf{seq}} : (m', t')\end{array}}{P, F, m_c \vdash s; s_{\mathsf{seq}} : (m', t')}$$