Separation Logic, Abstraction and Inheritance

Matthew J. Parkinson
University of Cambridge, UK
Matthew.Parkinson@cl.cam.ac.uk

Gavin M. Bierman
Microsoft Research Cambridge, UK
gmb@microsoft.com

Abstract

Inheritance is a fundamental concept in object-oriented programming, allowing new classes to be defined in terms of old classes. When used with care, inheritance is an essential tool for object-oriented programmers. Thus, for those interested in developing formal verification techniques, the treatment of inheritance is of paramount importance. Unfortunately, inheritance comes in a number of guises, all requiring subtle techniques.

To address these subtleties, most existing verification methodologies typically adopt one of two restrictions to handle inheritance: either (1) they prevent a derived class from restricting the behaviour of its base class (typically by syntactic means) to trivialize the proof obligations; or (2) they allow a derived class to restrict the behaviour of its base class, but require that every inherited method must be reverified. Unfortunately, this means that typical inheritance-rich code either cannot be verified or results in an unreasonable number of proof obligations.

In this paper, we develop a separation logic for a core object-oriented language. It allows derived classes which override the behaviour of their base class, yet supports the inheritance of methods without reverification where this is safe. For each method, we require two specifications: a *static* specification that is used to verify the implementation and direct method calls (in Java this would be with a super call); and a *dynamic* specification that is used for calls that are dynamically dispatched; along with a simple relationship between the two specifications. Only the dynamic specification is involved with behavioural subtyping. This simple separation of concerns leads to a powerful system that supports all forms of inheritance with low proof-obligation overheads. We both formalize our methodology and demonstrate its power with a series of inheritance examples.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Program Verification—class invariants; D.3.3 [*Programming Languages*]: Language Constructs and Features—Classes and inheritance

General Terms Languages, Theory, Verification

Keywords Separation Logic, Modularity, Classes

1. Introduction

1.1 Motivation

Inheritance is a fundamental concept in object-oriented programming. It allows new classes to be defined in terms of existing classes. These new, or derived, classes inherit both attributes and behaviour from their base classes. There are several different uses of inheritance in object-oriented code:

Specialization: One common use of inheritance is to create a specialization of the base class. This typically involves directly inheriting members from the base class and extending this set with further members.

Overriding: Most object-oriented languages allow a class to replace some of the members of the base class, these members are said to override the definitions in the base class.

Code re-use: Sometimes inheritance is used purely for code re-use, that is, a derived class is not intended to be used in the same places as its base class, but rather they just share code.

Our main concern is providing practical, modular verification methodologies to enable programmers to reason about and document their code. As inheritance—in all its guises—plays such an important role in object-oriented code, we contend that any verification methodology must provide practicable techniques for dealing with it. In other words, the programmer should be able to verify common inheritance patterns without substantial rewriting of their code. Moreover, these typical patterns should not result in unreasonable proof obligations on the programmer.

Let us consider some examples of these uses of inheritance and the problems they raise for formal verification. In this paper we only address languages that support *single* inheritance; for example, languages such as C$^\sharp$ and Java.[1] In Figure 1, we define a base class, Cell, and a derived class, Recell. This code uses both specialization and overriding. We will consider DCell and code re-use later. The base class has a field val, and two methods, set and get. The derived class directly inherits the field val, and method get. In addition it specializes its base class by defining a new field bak; and also overrides the set method. (The C$^\sharp$ expression **base**.m accesses the m member of the base class of the current object. It is written super.m in Java.)[2]

Firstly, by defining the derived class, we are able to pass Recell objects as if they were Cell objects. This syntactic property is guaranteed by the type system. The corresponding semantic property, known as substitutivity, is that whenever an object of type Cell is

[1] In this paper, we shall write our code examples in a C$^\sharp$-style syntax, but our techniques apply to Java, Visual Basic and other single-inheritance object-oriented languages.

[2] The overridden set method is very abstract in using a **base** call for get, rather than direct field access or dynamic dispatch. We discuss the alternatives in §5.5.

```
class Cell
{
  public int val;

  public virtual void set(int x)
  {
     this.val=x;
  }

  public virtual int get()
  {
     return this.val;
  }
}
```

```
class Recell: Cell
{
  public int bak;
  public override void set(int x)
  {
     this.bak = base.get();
     base.set(x);
  }
}
class DCell:Cell
{
  public override void set(int x)
  { base.set(2*x); }
}
```

Figure 1. Examples of inheritance.

expected, supplying an object of type Recell will not change the behaviour of the program. Liskov and Wing (1994) defined a notion called *behavioural subtyping* that guarantees the property of substitutivity.

Secondly, Recell inherits the Cell's body for the get method. This is correct at the level of types, but is it semantically valid to inherit this method? Unfortunately, this is a non-trivial problem. To simplify matters, most current verification methodologies adopt one of two restrictions: either they (1) prevent a derived class restricting the behaviour of the base class which trivializes the proof obligations, e.g. Müller (2002); Barnett et al. (2004), or (2) they allow a derived class to restrict the behaviour of its base class, but require that all inherited methods are reverified (Parkinson and Bierman 2005). Neither of these approaches are satisfactory and one of the aims of this work was to remove these restrictions.

Now let us consider code reuse and the DCell class in Figure 1. As far as we are aware, most systems (for example, Barnett et al. (2005); Müller (2002)) cannot cope with this use of inheritance. The intention is that instances of class DCell always store double the value they have been set to. This use of inheritance is quite subtle; we have declared class DCell as a derived class essentially to enable us to inherit the get method code. However, it is clear that instances of DCell behave quite differently from instances of Cell, which is at odds with our assumptions of behavioural subtyping. (It's an instance of the "inheritance is not subtyping" phenomenon (Cook et al. 1990).)

Whilst such programming techniques are obviously fragile, it is our experience that this use of inheritance is quite common in the developer community. So, we aim to verify such uses of inheritance.

To summarize, our overall intention is to provide a flexible framework to allow programmers to formally specify and verify the behaviour of their object-oriented code. In this paper we concentrate specifically on verifying code that uses inheritance: where a class can re-use, extend or alter the representation and operations of its base class. We also impose a minimal set of requirements for any solution to this problem:

Soundness: We insist that our solution is sound, by which we mean that a verified program will satisfy its specification.

Modular: We insist that our solution is modular, by which we mean that when new components are added to the system, no old component needs to be re-specified or re-verified.

No base class code required: An inherited method need never be reverified, or, alternatively, a class need not see the code of its base class (Ruby and Leavens 2000).

Breadth: This is a harder criteria to quantify, but we insist that our approach can verify the typical patterns of inheritance use in real-world software. In this case, we wish to support specialization, overriding and code re-use.

1.2 Our proposal

Our proposal for supporting inheritance builds on our earlier work (Parkinson and Bierman 2005) that developed a separation logic for reasoning about object-oriented code. Separation logic offers a particularly good framework as it supports local reasoning about stateful computation (Reynolds 2002) and hence deals directly with issues of ownership and state modification (in other systems these require additional complications to the underlying framework).

In earlier work (Parkinson and Bierman 2005) we proposed the notion of *abstract predicate families* to deal with the simple case of inheritance where every method is either overridden, or reverified in the derived class. Previously, when we verified a method body $\bar{\mathsf{s}}$ for class C with pre-condition P and post-condition Q, we verified the following.

$$\{P \wedge \textbf{this}{:}\mathsf{C}\}\ \bar{\mathsf{s}}\ \{Q\}$$

The type information, $\textbf{this}{:}\mathsf{C}$, enables the use of abstract predicate families, but prevents inheritance of methods without reverification: the verification is specific to a single class.

In this work, we provide a generalized logic that allows these restrictions to be lifted. For each method, we require two specifications: a *static* specification, that is used to verify the implementation and direct method calls (in Java this would be with a super call); and a *dynamic* specification, that is used for calls that are dynamically dispatched; along with a simple relationship between the two specifications. Only the dynamic specification is involved with behavioural subtyping.

We will demonstrate the use of dynamic and static specifications by example, but first we must recap some details of abstract predicates and abstract predicate families. An abstract predicate has a name, a definition, and a scope. Within the scope one can freely swap between using the abstract predicate's name and its definition, but outside its scope it must be handled atomically, i.e. by its name. Thus the scope defines the abstraction boundary for the abstract predicate. Whilst this handles simple modules, it is not powerful enough to deal with object-oriented abstraction, as we wish each class to be able to provide its own definition of the predicate.

To deal with this, we introduced the notion of an abstract predicate family. Informally, it can be seen as a *dynamically dispatched* predicate. Just as the code for a method invocation is chosen based on the dynamic type of the instance parameter, we mirror this dispatch behaviour in the logic. An abstract predicate family uses its first argument to choose the definition of the predicate, i.e. if the first argument is of type C then the definition of the abstract predicate family is the one that class C defines:

$$x : \mathsf{C} \Rightarrow (\alpha(x, \overline{y}) \Leftrightarrow \alpha_{\mathsf{C}}(x, \overline{y}))$$

where α is an abstract predicate family, and α_{C} is the definition for class C.

Returning to our Cell example, we could define an abstract predicate family $\mathit{Val}(x, v)$, which for the Cell class is defined as $x.\mathsf{val} \mapsto v$, that is when verifying the Cell class we assume:

$$x : \mathsf{Cell} \Rightarrow (\,\mathit{Val}(x, v) \Leftrightarrow x.\mathsf{val} \mapsto v)$$

Other classes are free to define their own entry for the Val family. This only specifies the definition for the Cell class; all other classes are unspecified, hence the proof is independent of their definition.

Now let us specify the Cell class. We will give the dynamic specification using abstract predicate families to allow more behavioural subtypes, and the static specification will closely mirror the actual implementation:

```
class Cell
```

```
{
  public int val;

  public virtual void set(int x)
  dynamic { Val(this, _)}_{ Val(this, x)}
  static {this.val ↦ _}_{this.val ↦ x)}
  { ... }

  public virtual int get()
  dynamic { Val(this, x)}_{ Val(this, x) ∗ ret = x}
  static {this.val ↦ x}_{this.val ↦ x ∗ ret = x}
  { ... }
}
```

The static specifications describe precisely how the methods work, that is set modifies the val field to contain x; and get returns the value stored in the val field.[3] The dynamic specification is given in terms of the *Val* abstract predicate family to enable derived classes to alter the behaviour.

If the dynamic specification was given in terms of the fields accessed then a derived class would not be able to extend or alter the behaviour in any way (Leino 1998). Similarly if the static specification was given in terms of the abstract predicate family, then the derived class would not be able to inherit the method without knowing the hidden representation (Parkinson and Bierman 2005). By providing both a static and dynamic specification, we can inherit methods without reverification, and this also allows derived classes to alter the representation and behaviour if they override methods.

Let us return to the derived class DCell. We wish to provide a specification that allows it to be a considered a behavioural subtype of Cell in spite of its radically different behaviour. The DCell class must behave the same as its parent's dynamic specification. If we define the *Val* predicate family as *false* for the DCell then this proof obligation becomes trivial.

$$x : \mathsf{DCell} \Rightarrow (\mathit{Val}(x, v) \Leftrightarrow \mathit{false})$$

DCell ensures no client will ever have a *Val* predicate for a DCell. Therefore, in the "*Val*-world", DCell is not a subtype of Cell (that is, a variable of static type Cell that satisfies *Val* will not point to a DCell object).

```
class DCell : Cell{
   public override void set(int x)
   dynamic { Val(this, _)}_{ Val(this, x)}
     also {DVal(this, _)}_{DVal(this, x ∗ 2)}
   {...}

   public inherit int get()
   dynamic { Val(this, v)}_{ Val(this, v) ∗ ret = v}
     also {DVal(this, v)}_{DVal(this, v) ∗ ret = v}
}
```

Here we use **also** to mean satisfies both specifications, and **inherit** to provide a new specification for an inherited method. We introduce a new predicate family *DVal*, which defines the DCell's behaviour. We specify *DVal* as[4]

$$x : \mathsf{DCell} \Rightarrow (\mathit{DVal}(x, v) \Leftrightarrow x.\mathsf{val} \mapsto v)$$

[3] We could make the static specification more abstract to prevent a derived class depending on the precise representations of its base class. We could introduce a new predicate $\mathit{Val}_{\mathsf{Cell}}(x, v)$ for the entry in the *Val* family:

$$x : \mathsf{Cell} \Rightarrow (\mathit{Val}(x, v) \Leftrightarrow \mathit{Val}_{\mathsf{Cell}}(x, v))$$

This prevents the derived class depending on the unknown definition of the Cell class. As, we do in the rest of the paper.

[4] We could alternatively specify it as

$$x : \mathsf{DCell} \Rightarrow (\mathit{DVal}(x, v) \Leftrightarrow \mathit{Val}_{\mathsf{Cell}}(x, v))$$

to be abstract in the Cell's representation.

The DCell does satisfy the specification of Cell, but only vacuously. However, clients do not need to know the specification is only vacuously satisfied. They will never be able to observe this.

1.3 Contributions and content

This paper contains a number of novel contributions to the field of object-oriented verification. We explore the power of our system by considering a number of examples that exhibit typical uses of inheritance. Many of these examples are not supported by existing techniques. (Some more direct comparisons are given in §6.)

More specifically, the main contributions of this work are as follows.

- The separation of method specifications into "static" and "dynamic" specifications,
- The formalization of the proof obligations resulting from this separation,
- An elegant generalization of the formalization of abstract predicate families based on higher-order separation logic, and
- A systematic exploration of the expressive power of our logic by considering a number of typical programs exploiting various aspects of inheritance.

The proof system defined in this paper is modular, and does not require a derived class to see the code of its base class to verify its method. It can support many uses of inheritance: where a derived class extends its base class, where it restricts the behaviour of its base class, where it changes the behaviour of its base class, and even where it changes the representation of its base class. No other proof system that we are aware of can handle all of these uses of inheritance.

The rest of the paper is structured as follows. In §2 we define a core object-oriented language with annotated method definitions. In §3 we define formally our proof system, based on separation logic. In §4 we show how to simplify the annotations. In §5 we verify a number of example uses of inheritance. We conclude in §6 by comparing our proof system to others. In Appendix A we give an overview of the semantics of our proof system.

2. A programming language with specifications

In this section we define formally both the core object-oriented language we verify and the associated annotations.

2.1 Syntax

Our core language is an extended, featherweight fragment of C$^\sharp$ called FVC$^\sharp$ (for Featherweight Verified C$^\sharp$), that is similar to various formalized fragments of Java (Flatt et al. 1997; Bierman et al. 2004). The main extension to C$^\sharp$ is that we annotate method definitions with static and dynamic specifications. The syntax of FVC$^\sharp$ class definitions, method definitions, and statements is defined as follows.

FVC$^\sharp$ programs

L ::= **class** C : D{ **public** $\overline{\mathsf{T}}$ $\overline{\mathsf{f}}$; $\overline{\mathsf{A}}$ K $\overline{\mathsf{M}}$ }	Class definitions
A ::= **define** $\alpha_{\mathsf{C}}(\overline{x})$ **as** P	Predicate Family Entry
K ::= **public** C() Sd Ss {$\overline{\mathsf{s}}$}	Constructor
M ::=	Method definition
public virtual C m($\overline{\mathsf{D}}$ $\overline{\mathsf{x}}$) Sd Ss B	Virtual method
public override C m($\overline{\mathsf{D}}$ $\overline{\mathsf{x}}$) Sd Ss B	Overridden method
public inherit C m($\overline{\mathsf{D}}$ $\overline{\mathsf{x}}$) Sd Ss;	Inherited method
Sd ::= **dynamic** S	Dynamic specification

Ss ::= **static** S	Static specification
S ::=	Method specification
$\{P\}$-$\{Q\}$	
S **also** $\{P\}$-$\{Q\}$	
B ::= { $\overline{\mathsf{C}}\ \overline{\mathsf{x}}$; $\overline{\mathsf{s}}$ **return** y; }	Method body
s ::=	Statement
x = y;	Assignment
x = **null**;	Initialization
x = y.f;	Field access
x.f = y;	Field update
x = y.m($\overline{\mathsf{z}}$);	Dynamic method invocation
x = y.C::m($\overline{\mathsf{z}}$);	Direct method invocation
x = (C)y	Cast
if(x == y){$\overline{\mathsf{s}}$} **else** {$\overline{\mathsf{t}}$}	Equality test
x = **new** C();	Object creation

In the syntax rules we assume a number of metavariables: f ranges over field names, C,D over class names, m over method names, and x, y, z over program variables. We assume that the set of program variables includes a designated variable **this**, which cannot be used as an argument to a method (this restriction is imposed by the typing rules). We follow Featherweight Java, or FJ (Igarashi et al. 2001), and use an 'overbar' notation to denote sequences.

As with FJ, for simplicity we do not include any primitive types in FVC$^\sharp$, and we assume that there is a distinguished class Object that is at the root of the inheritance hierarchy. We do not formalize the type system of FVC$^\sharp$ here as it is entirely standard.

A FVC$^\sharp$ class definition, L, contains a collection of fields and method definitions and, for simplicity, a single constructor. A field is defined by a type and a name. A virtual method definition, M, is defined by a return type, a method name, an ordered list of arguments—where an argument is a variable name and a type—a method specification, S, and a method body, B.

A method specification consists of a dynamic specification, Sd, and a static specification, Ss, each consisting of a sequence of pre- and post-conditions separated by **also**. The use of these was informally presented in the previous section, and the formal conditions on their use is given later. In §4 we show how one can drop one or the other specification, but in our featherweight language we insist on both specifications to help in the definitions.

We also insist that all inherited methods are explicitly specified in the derived class. This is partly to simplify some definitions, but also to allow derived classes to provide new specifications for inherited methods. Clearly, outside the formalization we would not insist on specifying inherited methods—in which case it would be assumed that the method specifications were inherited also.

A method body, B, consists of a number of local variable declarations, followed by a sequence of statements and a **return** statement. The real economy of FVC$^\sharp$ is that we do not have any syntactic forms for expressions (or even promotable expressions (Bierman et al. 2004)), and that the forms for statements are syntactically restricted. All expression forms appear only on the right-hand side of assignments. Moreover expressions only ever involve variables. In this respect, our form for statements is reminiscent of the A-normal form for λ-terms (Flanagan et al. 1993). A statement, s, is either an assignment, a field access, a field update, a method invocation, a cast, a conditional, or an object creation. In addition, we support direct method invocations using a C++-style syntax, e.g. c.C::m(y). This syntax subsumes the standard C$^\sharp$ **base** calls: in a class derived from a base class B, the statement x=**base**.m(y) in C$^\sharp$ is written as x=**this**.B::m(y) in FVC$^\sharp$.

In FVC$^\sharp$ we follow FJ and simplify matters by not considering overloading of methods or constructor methods. In spite of the heavy syntactic restrictions, we have not lost any expressivity; it is quite simple to translate a more conventional calculus with expressions and promotable expressions into FVC$^\sharp$. Another advantage of our approach is that we have no need for the 'stupid' rules of FJ.

In FVC$^\sharp$ we assume a rather large amount of syntactic regularity to make the definitions compact. All class definitions must (1) include a supertype; (2) start with all the declarations of the variables local to the method (hence a method block is a sequence of local variable declarations, followed by a sequence of statements); (3) have a **return** statement at the end of every method; and (4) write out field accesses explicitly, even when the receiver is **this**.

2.2 Dynamic semantics

The dynamic semantics of FVC$^\sharp$ are routine and are omitted. However, it is interesting to compare the reduction rules for the two forms of method invocations. These are as follows.

$$\frac{\begin{array}{c} \mathit{mbody}(\mathsf{C},\mathsf{m}) = (\overline{\mathsf{z}}'',\mathsf{B}) \\ \mathsf{B} \equiv \overline{\mathsf{C}}\ \overline{\mathsf{x}};\ \overline{\mathsf{s}}'\ \textbf{return}\ \mathsf{x}'; \qquad \theta = [\mathsf{y}_1,\overline{\mathsf{z}}',\overline{\mathsf{x}}'/\textbf{this},\overline{\mathsf{z}}'',\overline{\mathsf{x}}] \\ \overline{\mathsf{z}}',\overline{\mathsf{x}}'\ \text{fresh} \qquad S' = S[\overline{\mathsf{z}' \mapsto S(\mathsf{z})}] \end{array}}{S, H, \mathsf{y}_0\ =\ \mathsf{y}_1.\mathsf{C}::\mathsf{m}(\overline{\mathsf{z}});\overline{\mathsf{s}} \longrightarrow S', H, (\theta\overline{\mathsf{s}}')\mathsf{y}_0\ =\ (\theta\mathsf{x}');\ \overline{\mathsf{s}}}$$

$$\frac{\mathit{type}(H(S(\mathsf{y}))) = \mathsf{C} \qquad S, H, \mathsf{x} = \mathsf{y}.\mathsf{C}::\mathsf{m}(\overline{\mathsf{z}});\ \overline{\mathsf{s}} \longrightarrow S', H', \overline{\mathsf{s}}'}{S, H, \mathsf{x}\ =\ \mathsf{y}.\mathsf{m}(\overline{\mathsf{z}});\ \overline{\mathsf{s}} \longrightarrow S', H', \overline{\mathsf{s}}'}$$

We define the dynamic semantics in terms of transitions between configurations. A configuration is a triple $S, H, \overline{\mathsf{s}}$ consisting of (1) a stack S, which is a map from identifiers to heap addresses; (2) a heap H, which is a map from heap addresses to object representations, where an object representation has a type and a map from field name to addresses; and (3) a sequence of statements, $\overline{\mathsf{s}}$, which is the program being evaluated.

The interesting feature of the (dynamic dispatch) method invocation transition rule is its use of a direct method invocation.

3. Formalizing the proof system

In this section we formalize the proof system for reasoning about FVC$^\sharp$ programs. An overview of the semantics of this system is given in Appendix A.

3.1 Logic syntax

In this subsection we define the fragment of separation logic that we use to reason about our FVC$^\sharp$ programs. Formulae, P are defined by the following grammar, where x, y, z ranges over variable names, α ranges over predicates. We encode the rest of the usual connectives.

Formulae

$$\begin{array}{rcl} P, Q & ::= & \forall x.P \mid P \Rightarrow Q \mid \mathit{false} \mid \alpha(\overline{x}) \mid e = e' \mid x : \mathsf{C} \\ & \mid & x.f \mapsto e \mid P * Q \mid P -\!\!* Q \end{array}$$

$$e ::= x \mid \textbf{null}$$

Separation logic (Ishtiaq and O'Hearn 2001; O'Hearn et al. 2001; Reynolds 2002) is an extension to Hoare logic that permits reasoning about shared mutable state. It extends Hoare logic by adding spatial connectives to the assertion language, which allow us to assert that two portions of the heap are disjoint, that is $P * Q$ means the heap can be split into two disjoint parts in which P and Q hold respectively. Space prevents us from giving a more thorough introduction; the reader is referred to the tutorial by Reynolds (2002) for a general introduction, and to Parkinson (2005) for an introduction to the use of separation logic for verifying object-oriented programs.

We define an environment, Γ, that contains the static and dynamic specifications specified in a $\mathsf{FVC}^\sharp$ program. We write $\mathsf{C.m}(\overline{\mathsf{x}})\colon \{P\}\_\{Q\} \in \Gamma$ to denote that Γ contains the dynamic specification $\{P\}\_\{Q\}$ that is associated with the method m with parameters $\overline{\mathsf{x}}$ defined in class C. Similarly $\mathsf{C{::}m}(\overline{\mathsf{x}})\colon \{S\}\_\{T\} \in \Gamma$ denotes that Γ contains the static specification $\{S\}\_\{T\}$ that is associated with the method m with parameters $\overline{\mathsf{x}}$ defined in class C. We assume two special specifications $\mathsf{C..ctor}()$ and $\mathsf{C{::}.ctor}()$ for the static and dynamic specifications of the constructor of class C.

We also need to define an environment, Δ, that stores the abstract predicate families and their definitions. In previous work (Parkinson and Bierman 2005), we defined this as a set of predicate definitions, along with some complicated rules for controlling its use. One contribution of this paper is to utilize some observations originating from more recent work on higher-order separation logic (Biering et al. 2007), to simplify this representation. We give the details in §3.2, but for now the reader can just consider Δ as containing the abstract predicate families and their definitions.

The judgements for reasoning about $\mathsf{FVC}^\sharp$ programs are of the form $\Delta; \Gamma \vdash \{P\}\overline{\mathsf{s}}\{Q\}$, meaning that given environments Γ and Δ, the statement sequence $\overline{\mathsf{s}}$ satisfies the specification $\{P\}\_\{Q\}$.

The axioms and rules for forming valid judgements can be divided into "structural rules" and "program rules". The structural rules are those that work independently of the programs, i.e. they manipulate purely the pre- and post-conditions. These have been given elsewhere for separation logic (Parkinson 2005), so we do not repeat them here except for the "frame rule", as it is crucial to the "local reasoning" principle of separation logic. It is defined as follows.

$$\frac{\Delta; \Gamma \vdash \{P\}\overline{\mathsf{s}}\{Q\}}{\Delta; \Gamma \vdash \{P * R\}\overline{\mathsf{s}}\{Q * R\}}$$

The program rules for forming valid judgements have also been given elsewhere. However, we give below the rules for both forms of method invocation. First, the rule for dynamic dispatch invocation, which is as follows.[5]

$$\frac{x \text{ has static type } \mathsf{C} \qquad \mathsf{C.m}(\overline{\mathsf{x}})\colon \{P\}\_\{Q\} \in \Gamma}{\begin{array}{c}\Delta; \Gamma \vdash \{P[x, \overline{y}/\textbf{this}, \overline{x}] \wedge \textbf{this} \neq \textbf{null}\} \\ \mathsf{z} = \mathsf{x.m}(\overline{\mathsf{y}}) \\ \{Q[z, x, \overline{y}/ret, \textbf{this}, \overline{x}]\}\end{array}}$$

The rule for direct method calls is similar, except that the static specification is used.

$$\frac{\mathsf{C{::}m}(\overline{\mathsf{x}})\colon \{S\}\_\{T\} \in \Gamma}{\begin{array}{c}\Delta; \Gamma \vdash \{S[x, \overline{y}/\textbf{this}, \overline{x}] \wedge \textbf{this} \neq \textbf{null}\} \\ \mathsf{z} = \mathsf{x.C{::}m}(\overline{\mathsf{y}}) \\ \{T[z, x, \overline{y}/ret, \textbf{this}, \overline{x}]\}\end{array}}$$

Finally, we give the rule for constructing a new object.

$$\frac{\mathsf{C..ctor}()\colon \{P\}\_\{Q\} \in \Gamma}{\Delta; \Gamma \vdash \{P\}\mathsf{x} = \textbf{new } \mathsf{C}()\{Q[\mathsf{x}/\textbf{this}]\}}$$

3.2 Abstract predicate families

Next we explain how we represent abstract predicate families in this framework (Parkinson and Bierman 2005). Rather than presenting a syntax for a context and rules for manipulating and using this context, we simply define our context, Δ, as a conjunction of formulae from our logic:

$$\Delta ::= P \mid \Delta_1 \wedge \Delta_2 \mid \exists \alpha.\Delta$$

The existential is used to hide redundant definitions.

[5] To simplify the presentation, we assume that the return variable, z, is not an argument or the instance parameter (receiver). This restriction can be satisfied trivially by adding an additional assignment to a fresh variable z' before the rule and replacing the uses of z by z': $\mathsf{z}' = \mathsf{z}; \mathsf{z} = \mathsf{x}[\mathsf{z}'/\mathsf{z}].\mathsf{m}(\overline{\mathsf{y}}[\mathsf{z}'/\mathsf{z}])$

By using a formula from the logic the work can be connected with developments in higher-order separation logic (Biering et al. 2007; Nanevski et al. 2007; Krishnaswami et al. 2007). To simplify the presentation we do not use the full generality of higher-order separation logic in this paper: we simply require a second-order quantifier.

Now let us consider representing predicate definitions in this way. We might wish to define a predicate by **define** $Point(x, v)$ **as** $x.\mathsf{f} \mapsto v$, this can be seen as saying the following formula is true:

$$\forall x, v.\ Point(x, v) \Leftrightarrow x.\mathsf{f} \mapsto v$$

However, this is not powerful enough to reason about object-oriented abstractions. Due to dynamic dispatch, a method call is chosen based on the dynamic type of the instance parameter. Here we mirror this in the logic as the predicate definition is chosen by the dynamic type of the first parameter. So we could say **define** $Point_{\mathsf{C}}(x, v)$ **as** $x.\mathsf{f} \mapsto v$, which would mean two things: (1) the *entry* for the abstract predicate family $Point$ is $Point_{\mathsf{C}}$; and (2) this entry is defined as $x.\mathsf{f} \mapsto v$. These statements can be provided by the following

$$\begin{array}{l}(\forall x, v.\ Point_{\mathsf{C}}(x, v) \Leftrightarrow x.\mathsf{f} \mapsto v) \\ \quad \wedge (\forall x, v.\ x : \mathsf{C} \Rightarrow (Point_{\mathsf{C}}(x, v) \Leftrightarrow Point(x, v)))\end{array}$$

As a naming convention, we will use names, α, without a subscript to represent an abstract predicate *family*, and with α_C to represent the *entry* for class C in family α.

The role of the predicate arguments can be seen as analogous to the use of model fields in other work (Leino and Müller 2006). Derived classes generally introduce more model fields, hence we wish to interpret a predicate at many arities. To encode model fields more directly we could pass a record rather than a list of arguments. The implication below would then correspond to width subtyping and forgotten fields are existentially quantified. Returning to our point example, we might wish to be able to forget the outer most argument as a base class does not use this parameter:

$$Point(x) \Leftrightarrow \exists v.\ Point(x, v)$$

We can now define the formal translation of a definition to a logical assumption. We break the translation into three parts: (1) family to entry, $FtoE(\alpha, \mathsf{C})$ is a formula that connects the family, α, with the entry α_{C} assuming the first argument is of type C; (2) entry to definition, $EtoD(\textbf{define } \alpha_{\mathsf{C}}(x, \overline{x}) \textbf{ as } P)$ is a formula that connects the entry α_{C} with the definition P; and (3) changing arity, $A(\alpha; n)$ defines that family α can be given any arity less than n and the missing values are existentially quantified.

$$\begin{array}{l}FtoE(\alpha, \mathsf{C}) \stackrel{\text{def}}{=} \forall x, \overline{x}.\ x : \mathsf{C} \Rightarrow (\alpha(x, \overline{x}) \Leftrightarrow \alpha_{\mathsf{C}}(x, \overline{x})) \\ EtoD(\textbf{define } \alpha_{\mathsf{C}}(x, \overline{x}) \textbf{ as } P) \\ \qquad \stackrel{\text{def}}{=} \forall x, \overline{x}.\ \alpha_{\mathsf{C}}(x, \overline{x}) \Leftrightarrow P \\ A(\alpha; n+1) \stackrel{\text{def}}{=} A(\alpha; n) \wedge \\ \qquad \forall y_1, \ldots, y_n.\ \alpha(y_1, \ldots, y_n) \Leftrightarrow \exists z.\ \alpha(y_1, \ldots, y_n, z) \\ \quad A(\alpha; 0) \stackrel{\text{def}}{=} \text{true}\end{array}$$

We can translate an entry definition as (1) the formula connecting the family with the entry; (2) the formula connecting the entry with the definition; and (3) the formula specifying that the arity can be reduced.

$$\begin{array}{l}apf_{\mathsf{C}}(\textbf{define } \alpha_{\mathsf{C}}(x, \overline{x}) \textbf{ as } P) \\ \quad \stackrel{\text{def}}{=} FtoE(\alpha, \mathsf{C}) \wedge EtoD(\textbf{define } \alpha_{\mathsf{C}}(x, \overline{x}) \textbf{ as } P) \wedge A(\alpha; |\overline{x}|) \\ apf(\textbf{class } \mathsf{C}\colon \mathsf{D}\{\ \textbf{public } \overline{\mathsf{T}}\ \overline{\mathsf{f}};\ \mathsf{A}_1 \cdots \mathsf{A}_n\ \mathsf{K}\ \overline{\mathsf{M}}\ \}) \\ \quad \stackrel{\text{def}}{=} apf_{\mathsf{C}}(\mathsf{A}_1) \wedge \cdots \wedge apf_{\mathsf{C}}(\mathsf{A}_n)\end{array}$$

We have two rules to reason about these assumptions. The first allows us to strengthen our assumptions. If we can prove our program with just the assumptions Δ, then we can prove it with the weaker assumptions Δ'.

$$\frac{\Delta' \Rightarrow \Delta \qquad \Delta;\Gamma \vdash \{P\}\bar{s}\{Q\}}{\Delta';\Gamma \vdash \{P\}\bar{s}\{Q\}}\ \text{P-Weak}$$

The second proof rule allows the removal of predicates: this is the second-order analogy of the logical/ghost/auxilliary variable elimination rule.

$$\frac{\alpha \notin \mathrm{FP}(P, Q, \Gamma) \qquad \Delta;\Gamma \vdash \{Q\}\bar{s}\{R\}}{(\exists\alpha.\ \Delta);\Gamma \vdash \{Q\}\bar{s}\{R\}}\ \text{P-Elim}$$

where $\mathrm{FP}(P, Q, \Gamma)$ is the set of free predicate names in P, Q and Γ.

Finally, the rule of consequence is modified to take account of the environment Δ.

$$\frac{\Delta \Rightarrow (P \Rightarrow P') \qquad \Delta;\Gamma \vdash \{P'\}\bar{s}\{Q'\} \qquad \Delta \Rightarrow (Q' \Rightarrow Q)}{\Delta;\Gamma \vdash \{P\}\bar{s}\{Q\}}$$

3.3 Refinement and behavioural subtyping

A number of authors have offered definitions of behavioural subtyping, but in this work we follow Leavens and Naumann (2006) and propose a formulation in terms of a natural refinement order on specifications. We say that a specification $\{P_2\}\_\{Q_2\}$ refines another specification $\{P_1\}\_\{Q_1\}$, if for all programs $\bar{s}$, if $\bar{s}$ satisfies the latter specification, then it also satisfies the former.

We characterize[6] specification refinement using the structural rules of Hoare and Separation logic (Consequence, Frame, Logical[7] variable elimination); that is, there exists a proof of the form

$$\frac{\dfrac{\Delta \vdash \{P_1\}\_\{Q_1\}}{\vdots}}{\Delta \vdash \{P_2\}\_\{Q_2\}}$$

When such a proof exists, we write $\Delta \vdash \{P_1\}\_\{Q_1\} \Longrightarrow \{P_2\}\_\{Q_2\}$. We often need to introduce some type information in specification refinement. To do so, we define

$$\Delta \vdash \{P_1\}\_\{Q_1\} \overset{\mathbf{this}:\,\mathsf{C}}{\Longrightarrow} \{P_2\}\_\{Q_2\}$$
$$\overset{\text{def}}{=} \quad \Delta \vdash \{P_1\}\_\{Q_1\} \Longrightarrow \{P_2 * \mathbf{this}\colon \mathsf{C}\}\_\{Q_2\}$$

We often need to combine specifications. This is written as **also**, and is encoded as follows using logical (auxiliary) variables.

Definition 1. $\{P_1\}\_\{Q_1\}$ $\mathbf{also}_X$ $\{P_2\}\_\{Q_2\}$ *is defined as* $\{(P_1 \wedge X{=}1) \vee (P_2 \wedge X{\neq}1)\}\_\{(Q_1 \wedge X{=}1) \vee (Q_2 \wedge X{\neq}1)\}$

We omit the X to mean selecting a fresh variable. In our verification rules, we omit **also** by encoding the specifications into a single specification.

Lemma 2.

1. $\Delta \vdash (\{P_1\}\_\{Q_1\}\ \mathbf{also}\ \{P_2\}\_\{Q_2\}) \Longrightarrow \{P_1\}\_\{Q_1\}$
2. $\Delta \vdash (\{P_1\}\_\{Q_1\}\ \mathbf{also}\ \{P_2\}\_\{Q_2\}) \Longrightarrow \{P_2\}\_\{Q_2\}$

3.3.1 Method verification

There are three forms of method definitions in $\mathsf{FVC}^\sharp$. A method can be defined (1) as **virtual**, if it is not defined in its base class; or (2) as **inherit**, if it is not defined in this class, but is defined in its base class; or (3) as **override**, if it is defined both by this class and its base class. For each of these types of method definitions, we will provide the appropriate verification rule. The judgement form is written as $\Delta;\Gamma \vdash \mathsf{M}\ \text{in}\ \mathsf{E}$, which means informally that "given environments Γ and Δ, the method definition M in class E can be verified to meet its specification." (In what follows we write $\mathsf{C} \prec_1 \mathsf{D}$ when class D is the immediate base class of derived class C, i.e. **class** C : D $\{\ldots\}$. Additionally, to simplify the presentation, we assume that methods do not modify the variables containing the arguments. Methods can be trivially rewritten to this form.)

[6] We currently do not have an adaption completeness result for our proof system, and so we cannot assert whether our syntactic characterization of refinement is complete. Yang's thesis provides an adaption completeness result for separation logic without object-oriented features (Yang 2001).

[7] Sometimes called auxiliary or ghost variables.

First, we define the rule for verifying a new **virtual** method.

$$\frac{\begin{array}{ll} \mathsf{B} = \{\ \bar{\mathsf{G}}\ \bar{\mathsf{y}};\ \bar{\mathsf{s}}\ \mathbf{return}\ \mathsf{z};\ \} & \\ \mathsf{Sd} = \mathbf{dynamic}\ \{P_{\mathsf{E}}\}\_\{Q_{\mathsf{E}}\} & \\ \mathsf{Ss} = \mathbf{static}\ \{S_{\mathsf{E}}\}\_\{T_{\mathsf{E}}\} & \\ \Delta \vdash \{S_{\mathsf{E}}\}\_\{T_{\mathsf{E}}\} \overset{\mathbf{this}:\,\mathsf{E}}{\Longrightarrow} \{P_{\mathsf{E}}\}\_\{Q_{\mathsf{E}}\} & \text{(Dynamic dispatch)} \\ \Delta;\Gamma \vdash \{S_{\mathsf{E}}\}\bar{\mathsf{s}}\{T_{\mathsf{E}}[\mathsf{z}/ret]\} & \text{(Body Verification)} \end{array}}{\Delta;\Gamma \vdash \mathbf{public\ virtual}\ \mathsf{C}\ \mathsf{m}(\bar{\mathsf{D}}\ \bar{\mathsf{x}})\ \mathsf{Sd}\ \mathsf{Ss}\ \mathsf{B}\ \text{in}\ \mathsf{E}}$$

In this case there are just two proof obligations: we must verify that (1) the method body meets its **static** specification, (Body Verification); and (2) using the dynamic specification is valid for dynamic dispatch, (Dynamic dispatch). This second proof obligation forces a relationship between the static and dynamic specifications of a method. It corresponds to showing that if the object has type E and the dynamic specification is satisfied by the client, then the method body will execute successfully. Notice that by using the static specification we do not have to verify the body against the dynamic specification.

Next, we define the verification rule for inheriting a method.

$$\frac{\begin{array}{ll} \mathsf{E} \prec_1 \mathsf{F} & \\ \mathsf{Sd} = \mathbf{dynamic}\ \{P_{\mathsf{E}}\}\_\{Q_{\mathsf{E}}\} & \\ \mathsf{Ss} = \mathbf{static}\ \{S_{\mathsf{E}}\}\_\{T_{\mathsf{E}}\} & \\ \mathsf{F.m}(\bar{\mathsf{x}})\colon \{P_{\mathsf{F}}\}\_\{Q_{\mathsf{F}}\} \in \Gamma & \\ \mathsf{F{::}m}(\bar{\mathsf{x}})\colon \{S_{\mathsf{F}}\}\_\{T_{\mathsf{F}}\} \in \Gamma & \\ \Delta \vdash \{P_{\mathsf{E}}\}\_\{Q_{\mathsf{E}}\} \Longrightarrow \{P_{\mathsf{F}}\}\_\{Q_{\mathsf{F}}\} & \text{(Behavioural Subtyping)} \\ \Delta \vdash \{S_{\mathsf{F}}\}\_\{T_{\mathsf{F}}\} \Longrightarrow \{S_{\mathsf{E}}\}\_\{T_{\mathsf{E}}\} & \text{(Inheritance)} \\ \Delta \vdash \{S_{\mathsf{E}}\}\_\{T_{\mathsf{E}}\} \overset{\mathbf{this}:\,\mathsf{E}}{\Longrightarrow} \{P_{\mathsf{E}}\}\_\{Q_{\mathsf{E}}\} & \text{(Dynamic dispatch)} \end{array}}{\Delta;\Gamma \vdash \mathbf{public\ inherit}\ \mathsf{C}\ \mathsf{m}(\bar{\mathsf{D}}\ \bar{\mathsf{x}})\ \mathsf{Sd}\ \mathsf{Ss};\ \text{in}\ \mathsf{E}}$$

In this case there are three proof obligations: we must verify that (1) the new dynamic specification is a valid behavioural subtype, (Behavioural Subtyping); (2) the method meets the static specification, (Inheritance); and (3) (as before) using the dynamic specification is valid for dynamic dispatch, (Dynamic dispatch). The first proof obligation amounts to requiring that whenever it is valid to use the dynamic specification of the base class, it is also valid to use the dynamic specification of this class, E. The second proof obligation amounts to showing that the inherited method body satisfies the new static specification. However, this rule does *not* use the inherited method body at all; it is *not* needed. The rule works purely at the level of the specifications.

Finally, we give the verification rule for overriding a method.

$$\frac{\begin{array}{ll} \mathsf{E} \prec_1 \mathsf{F} & \\ \mathsf{F.m}(\bar{\mathsf{x}})\colon \{P_{\mathsf{F}}\}\_\{Q_{\mathsf{F}}\} \in \Gamma & \\ \mathsf{B} = \{\ \bar{\mathsf{G}}\ \bar{\mathsf{y}};\ \bar{\mathsf{s}}\ \mathbf{return}\ \mathsf{z};\ \} & \\ \mathsf{Sd} = \mathbf{dynamic}\ \{P_{\mathsf{E}}\}\_\{Q_{\mathsf{E}}\} & \\ \mathsf{Ss} = \mathbf{static}\ \{S_{\mathsf{E}}\}\_\{T_{\mathsf{E}}\} & \\ \Delta \vdash \{P_{\mathsf{E}}\}\_\{Q_{\mathsf{E}}\} \Longrightarrow \{P_{\mathsf{F}}\}\_\{Q_{\mathsf{F}}\} & \text{(Behavioural Subtyping)} \\ \Delta \vdash \{S_{\mathsf{E}}\}\_\{T_{\mathsf{E}}\} \overset{\mathbf{this}:\,\mathsf{E}}{\Longrightarrow} \{P_{\mathsf{E}}\}\_\{Q_{\mathsf{E}}\} & \text{(Dynamic dispatch)} \\ \Delta;\Gamma \vdash \{S_{\mathsf{E}}\}\bar{\mathsf{s}}\{T_{\mathsf{E}}[\mathsf{z}/ret]\} & \text{(Body Verification)} \end{array}}{\Delta;\Gamma \vdash \mathbf{public\ override}\ \mathsf{C}\ \mathsf{m}(\bar{\mathsf{D}}\ \bar{\mathsf{x}})\ \mathsf{Sd}\ \mathsf{Ss}\ \mathsf{B}\ \text{in}\ \mathsf{E}}$$

Again there are three proof obligations: we must verify that: (1) the new dynamic specification is a valid behavioural subtype, (Behavioural Subtyping); (2) the method body meets the static specification, (Body Verification); and (3) using the dynamic specification is valid for dynamic dispatch, (Dynamic dispatch). This verification is almost identical to the previous, but here we can verify the body of the method against the static specification as it is defined in this class, E.

The second verification rule has a degenerate (but common) form, when a derived class inherits a method from a base class but does *not* provide a new specification. In this case, the verification rule degenerates to the following.

$$\frac{\begin{array}{l}\mathsf{E} \prec_1 \mathsf{F}\\ \mathsf{F.m}(\overline{\mathsf{x}})\colon \{P_\mathsf{F}\}\_\{Q_\mathsf{F}\} \in \Gamma\\ \mathsf{F::m}(\overline{\mathsf{x}})\colon \{S_\mathsf{F}\}\_\{T_\mathsf{F}\} \in \Gamma\\ \Delta \vdash \{S_\mathsf{F}\}\_\{T_\mathsf{F}\} \overset{\mathbf{this};\,\mathsf{E}}{\Longrightarrow} \{P_\mathsf{F}\}\_\{Q_\mathsf{F}\} \quad (\text{Dynamic dispatch})\end{array}}{\Delta;\Gamma \vdash \mathbf{public\ inherit}\ \mathsf{C}\ \mathsf{m}(\overline{\mathsf{D}}\ \overline{\mathsf{x}});\ \mathsf{in}\ \mathsf{E}}$$

There is just one proof obligation, which amounts to verifying that if the object has type E and the dynamic specification of the base class is satisfied by the client, then the method body will satisfy the specification. Again, it is worth pointing out that this proof obligation is at the level of the specifications; we do not need the inherited method body from the base class.

Finally, we need a special verification rule for a constructor method definition. This is as follows.

$$\frac{\begin{array}{l}\mathsf{E} \prec_1 \mathsf{F}\\ \mathit{fields}(\mathsf{E}) = \mathsf{f}_1,\ldots,\mathsf{f}_n\\ \mathsf{F::.ctor}()\colon \{S_\mathsf{F}\}\_\{T_\mathsf{F}\} \in \Gamma\\ \mathsf{Sd} = \mathbf{dynamic}\ \{P_\mathsf{E}\}\_\{Q_\mathsf{E}\}\\ \mathsf{Ss} = \mathbf{static}\ \{S_\mathsf{E}\}\_\{T_\mathsf{E}\}\\ \Delta \vdash \{S_\mathsf{E}\}\_\{T_\mathsf{E}\} \overset{\mathbf{this};\,\mathsf{E}}{\Longrightarrow} \{P_\mathsf{E}\}\_\{Q_\mathsf{E}\}\\ \Delta;\Gamma \vdash \{T_\mathsf{F} * R * Fs\}\overline{\mathsf{s}}\{T_\mathsf{E}\} \quad (\text{Body Verification})\\ S_\mathsf{E} \Rightarrow S_\mathsf{F} * \mathit{true}\\ R \Leftrightarrow S_\mathsf{F} \mathrel{-\circledast} S_\mathsf{E}\\ Fs = \mathbf{this}.\mathsf{f}_1 \mapsto \_ * \ldots * \mathbf{this}.\mathsf{f}_n \mapsto \_\end{array}}{\Delta;\Gamma \vdash \mathbf{public}\ \mathsf{E}()\ \mathsf{Sd}\ \mathsf{Ss}\ \{\overline{\mathsf{s}}\}}$$

This rule is complicated by the implicit base call at the beginning of the constructor, that is, before the constructor body $\overline{\mathsf{s}}$ begins executing, the base class constructor is executed. When verifying the body (Body Verification) the pre-condition is composed of three things: (1) the post-condition of the base class constructor call, (2) a formula R which intuitively is the disjoint state required by the constructor (and hence is given by the formula $S_\mathsf{F} \mathrel{-\circledast} S_\mathsf{E}$[8]), and (3) a representation of the fields defined in E but not including the fields inherited from the base class.

3.4 Class verification

We can now use the method verification rules given above to verify a definition. The rule is as follows.

$$\frac{\forall \mathsf{M}_i \in \overline{\mathsf{M}}.\ \ \Delta;\Gamma \vdash \mathsf{M}_i\ \mathsf{in}\ \mathsf{C} \qquad \Delta;\Gamma \vdash \mathsf{K}}{\Delta;\Gamma \vdash \mathbf{class}\ \mathsf{C}\colon \mathsf{D}\{\ \mathbf{public}\ \overline{\mathsf{T}}\ \overline{\mathsf{f}};\ \overline{\mathsf{A}}\ \mathsf{K}\ \overline{\mathsf{M}}\ \}}$$

Informally, this means that to verify a class definition one must verify every method.

The rule for verifying a complete program is then as follows.

$$\frac{\begin{array}{l}\Gamma = \mathit{specs}(\mathsf{L}_1 \ldots \mathsf{L}_n)\\ \mathit{apf}(\mathsf{L}_1);\Gamma \vdash \mathsf{L}_1 \quad \cdots \quad \mathit{apf}(\mathsf{L}_n);\Gamma \vdash \mathsf{L}_n\\ \mathit{true};\Gamma \vdash \{\mathit{true}\}\overline{\mathsf{s}}\{\mathit{true}\}\end{array}}{\vdash \mathsf{L}_1 \ldots \mathsf{L}_n\overline{\mathsf{s}}}$$

Informally, this rule states that under the assumption that all the specifications are correct, every class definition must be verified,[9] along with verifying the main body, $\overline{\mathsf{s}}$.

Properties Given the proof rules above, we can now reconsider the criteria given in §1.1. First, our proof system is *sound*.

Theorem 3. *The program verification rule is sound. (See Appendix A for a formal statement of soundness and an overview of the proof.)*

[8] $P \mathrel{-\circledast} Q$ is defined as $\neg(P \mathrel{-\!\!*} \neg Q)$ and intuitively means subtracting P from Q.

[9] This assumption is valid as we are only dealing with partial correctness.

Secondly, our system is *modular*, i.e. the introduction of new methods or classes does not invalidate an existing proof. Thirdly, each method body is verified only once, even when defining an overridden method or inheriting a method from the base class.

Finally, in §5 we consider the applicability of our system by considering a number of typical uses of inheritance in object-oriented code.

4. Simplifying annotations

In many cases the dynamic and static specifications turn out to be very similar. Fortunately, there is a relatively simple process to derive the static specification from the dynamic, and vice versa.

Deriving static from dynamic We give a syntactic 'opening' function, $[\![P]\!]_C$, which opens all the abstract predicate families in P on the object **this** at type C. That is,

$$\begin{array}{rcll}
[\![\alpha(\mathbf{this},\overline{x})]\!]_C &\overset{\text{def}}{=}& \alpha_C(\mathbf{this},\overline{x}) &\\
[\![\alpha(y,\overline{x})]\!]_C &\overset{\text{def}}{=}& \alpha(y,\overline{x}) & \text{where } y \not\equiv \mathbf{this}\\
[\![\alpha_D(\overline{x})]\!]_C &\overset{\text{def}}{=}& \alpha_D(\overline{x}) &\\
[\![pr(\overline{x})]\!]_C &\overset{\text{def}}{=}& pr(\overline{x}) &\\
[\![\mathit{false}]\!]_C &\overset{\text{def}}{=}& \mathit{false} &\\
[\![P\ op\ Q]\!]_C &\overset{\text{def}}{=}& [\![P]\!]_C\ op\ [\![Q]\!]_C & \text{where } op ::= * \mid\Rightarrow\mid \mathrel{-\!\!*}\\
[\![\forall x.\ P]\!]_C &\overset{\text{def}}{=}& \forall x.\ [\![P]\!]_C &
\end{array}$$

where $pr(x, y)$ is either $x.f \mapsto y$, $x = y$ and $x : C$.

Lemma 4. $\mathbf{this} : C \Longrightarrow (P \Leftrightarrow [\![P]\!]_C)$

Hence, given a dynamic specification $\{P\}\_\{Q\}$, we can derive the static specification $\{[\![P]\!]_C\}\_\{[\![Q]\!]_C\}$, which automatically satisfies the (Dynamic dispatch) proof obligation.

Lemma 5. $\{[\![P]\!]_C\}\_\{[\![Q]\!]_C\} \overset{\mathbf{this}:C}{\Longrightarrow} \{P\}\_\{Q\}$

Proof.

$$\cfrac{\cfrac{\{[\![P]\!]_C\}\_\{[\![Q]\!]_C\}}{\{[\![P]\!]_C * \mathbf{this}{:}C\}\_\{[\![Q]\!]_C * \mathbf{this}{:}C\}}\ \text{Frame}}{\{P * \mathbf{this}{:}C\}\_\{Q\}}\ \text{Conseq}$$

□

Both the static specifications of the set and get methods of Cell can be inferred in this way.

Deriving dynamic from static If we only provide a static specification for a method, we assume the dynamic specification is identical to the static specification. This also satisfies the (Dynamic dispatch) proof obligation trivially.

5. Examples

In this section we give a number of examples to demonstrate the power and applicability of our proof system. All our examples involve inheritance of the Cell class that we described in §1. For completeness we give the complete definition of the Cell class, including the abstract predicate families and method specifications, in Figure 2.

Before we consider inheriting this class, we should first verify that it meets its own specification! For the three virtual methods, this means the two proof obligations, (Body Verification) and (Dynamic dispatch). Luckily, for all three methods the latter proof obligation is satisfied following Lemma 5. We give below a verification of the set method body, and suppress the verifications of get and swap.

$\{\mathit{Val}_{\mathsf{Cell}}(\mathbf{this}, \_)\}$
  $\{\mathbf{this}.\mathsf{val} \mapsto \_\}$**this**.val=x;$\{\mathbf{this}.\mathsf{val} \mapsto \mathsf{x}\}$
$\{\mathit{Val}_{\mathsf{Cell}}(\mathbf{this}, \mathsf{x})\}$

```
class Cell {
 int val;
 define $Val_{\mathsf{Cell}}(x, v)$ as $x$.val $\mapsto v$

 public Cell() dynamic $\{true\}_{\_}\{Val(\mathbf{this}, \_)\}$ {}

 public virtual void set(int x)
 dynamic $\{Val(\mathbf{this}, \_)\}_{\_}\{Val(\mathbf{this}, \mathsf{x})\}$
 {this.val=x;}

 public virtual int get()
 dynamic $\{Val(\mathbf{this}, v)\}_{\_}\{Val(\mathbf{this}, v) * ret = v\}$
 { return this.val; }

 public virtual void swap(Cell c)
 static $\{Val(\mathbf{this}, v_1) * Val(\mathsf{c}, v_2)\}_{\_}\{Val(\mathbf{this}, v_2) * Val(\mathsf{c}, v_1)\}$
 { int t,t2; t = this.get(); t2 = c.get(); this.set(t2); c.set(t); }
}
```

Figure 2. Source code for Cell examples

```
class Recell : Cell{
 int bak;

 define $Val_{\mathsf{Recell}}(x, v, o)$ as $Val_{\mathsf{Cell}}(x, v) * x$.bak $\mapsto o$

 public Recell () dynamic $\{true\}_{\_}\{Val(\mathbf{this}, \_, \_)\}$ {}

 public inherit int get()
 dynamic $\{Val(\mathbf{this}, v, o)\}_{\_}\{Val(\mathbf{this}, v, o) * ret = v\}$

 public override void set(int x)
 dynamic $\{Val(\mathbf{this}, v, \_)\}_{\_}\{Val(\mathbf{this}, \mathsf{x}, v)\}$
 { this.bak = this.Cell::get(); this.Cell::set(x); }

 public virtual void undo()
 dynamic $\{Val(\mathbf{this}, v, o)\}_{\_}\{Val(\mathbf{this}, o, \_)\}$
 { int tmp = this.bak; this.Cell::set(tmp); }
}
```

Figure 3. The Recell class

Before turning to deriving from this class, we consider in a little more detail the swap method. It is an example of the template method pattern (Gamma et al. 1994). It does not manipulate the data directly, but simply uses other methods to update the state. This allows the code to be reused in derived classes that alter the representation.

Hence it is interesting to consider the consequences of inheriting this method in some derived class, C, of Cell. If the derived class C inherits swap and does not alter its specification, then its only proof obligation is (Dynamic dispatch) which follows trivially (using the rule of consequence) as follows.

$$\frac{\{Val(\mathbf{this}, v_1) * Val(\mathsf{c}, v_2)\}_{\_}\{Val(\mathbf{this}, v_2) * Val(\mathsf{c}, v_1)\}}{\{Val(\mathbf{this}, v_1) * Val(\mathsf{c}, v_2) * \mathbf{this}{:}C\}_{\_}\{Val(\mathbf{this}, v_2) * Val(\mathsf{c}, v_1)\}}$$

Thus, any derived class is essentially free to inherit this method. It is particularly interesting to explore the consequences of different implementations of the swap method and the effects on the ability of derived classes to inherit this method. For example, consider if we had implemented swap using direct field access on the object, e.g.

```
public virtual void swap1(Cell c)
dynamic $\{Val(\mathbf{this}, v_1) * Val(\mathsf{c}, v_2)\}_{\_}\{Val(\mathbf{this}, v_2) * Val(\mathsf{c}, v_1)\}$
static $\{Val_{\mathsf{Cell}}(\mathbf{this}, v_1) * Val(\mathsf{c}, v_2)\}_{\_}\{Val_{\mathsf{Cell}}(\mathbf{this}, v_2) * Val(\mathsf{c}, v_1)\}$
{ int tmp = c.get(); c.set(this.val); this.val = tmp; }
```

The proof obligation to inherit this method would impose a constraint on any derived class. A better alternative would probably be to override the method.

A more optimised implementation of swap could directly access the fields of c as well.

```
public virtual void swap2(Cell c)
static $\{Val(\mathbf{this}, v_1) * Val(\mathsf{c}, v_2)\}_{\_}\{Val(\mathbf{this}, v_2) * Val(\mathsf{c}, v_1)\}$
{ int tmp = c.val; c.val = this.val; this.val = tmp; }
```

For the proof obligations to be satisfied, this would effectively impose a global constraint on all derived classes of Cell, that they preserve the usage of the val field:[10]

$$Val(x, v) \Leftrightarrow Val_{\mathsf{Cell}}(x, v) * ValLeft(x) \tag{1}$$

This kind of constraint is analogous to the condition in other work (Müller 2002) that derived class invariants cannot restrict the base class invariant. The derived class must define a meaning for the predicate family *ValLeft*. As we see later, this kind of constraint prevents many useful subtypes.

[10] Our program verification rules do not directly support this constraining of derived classes, but this could be trivially added.

5.1 Specialisation: Recell

Now let us consider Recell, a derived class of Cell, which additionally stores the previous value that was set to allow undo. The code is given in Figure 3. We consider the methods in turn: First, let us consider the get method. We must show that it is valid to inherit this method into the Recell class. First, we need to verify the proof obligation (Inheritance) i.e. to show that the static specification of the method in Recell refines the Cell static specification. This can be proved as follows.

$$\dfrac{\dfrac{\{Val_{\mathsf{Cell}}(x, v)\}_{\_}\{Val_{\mathsf{Cell}}(x, v) * ret{=}v\}}{\left\{\begin{array}{l}Val_{\mathsf{Cell}}(x, v)\\ * \, x.\mathsf{bak}{\mapsto}o\end{array}\right\}_{\_}\left\{\begin{array}{l}Val_{\mathsf{Cell}}(x, v)\\ * \, ret{=}v * x.\mathsf{bak}{\mapsto}o\end{array}\right\}}\,\text{Frame}}{\{Val_{\mathsf{Recell}}(x, v, o)\}_{\_}\{Val_{\mathsf{Recell}}(x, v, o) * ret{=}v\}}\,\text{Conseq}$$

This proof does *not* depend on either the internal representation of the Cell class or the body of the get method, only the static specification of the Cell class and the internal representation of the Recell class.

Second, we need to verify the proof obligation (Behavioural Subtyping), i.e. we must show that the Recell's dynamic specification of the get method is a valid behavioural subtype of the Cell's specification. This can be proved as follows.

$$\dfrac{\dfrac{\{Val(\mathbf{this}, v, o)\}_{\_}\{Val(\mathbf{this}, v, o) * ret{=}v\}}{\{\exists o.\ Val(\mathbf{this}, v, o)\}_{\_}\{\exists o.\ Val(\mathbf{this}, v, o) * ret{=}v\}}\,\text{VarElim}}{\{Val(\mathbf{this}, v)\}_{\_}\{Val(\mathbf{this}, v) * ret{=}o\}}\,\text{Conseq}$$

Note that this proof uses the arity manipulation described in §3.2.

Now, we turn our attention to the set method. The first proof obligation, (Body Verification), can be proved as follows.

```
$\{Val_{\mathsf{Recell}}(\mathbf{this}, v, \_)\}$
$\{Val_{\mathsf{Cell}}(\mathbf{this}, v) * \mathbf{this}.\mathsf{bak} \mapsto \_\}$
  tmp = this.Cell::get();
$\{Val_{\mathsf{Cell}}(\mathbf{this}, v) * \mathbf{this}.\mathsf{bak} \mapsto \_ * \mathsf{tmp} = v\}$
  this.bak = tmp;
$\{Val_{\mathsf{Cell}}(\mathbf{this}, v) * \mathbf{this}.\mathsf{bak} \mapsto v\}$
  this.Cell::set(x);
$\{Val_{\mathsf{Cell}}(\mathbf{this}, x) * \mathbf{this}.\mathsf{bak} \mapsto v\}$
$\{Val_{\mathsf{Recell}}(\mathbf{this}, x, v)\}$
```

```
class TCell : Cell {
 int val2;

 define $Val_{\mathsf{TCell}}(x, v)$ as $Val_{\mathsf{Cell}}(x, v) * x.\mathsf{val2} \mapsto v$

 TCell() dynamic $\{true\}_{\_}\{Val_{\mathsf{TCell}}(\mathbf{this}, \_)\}$ {}

 public override void set(int x)
 dynamic $\{Val(\mathbf{this}, \_)\}_{\_}\{Val(\mathbf{this}, \mathsf{x})\}$
 { this.val2=x; this.Cell::set(x); }

 public virtual void check()
 dynamic $\{Val(\mathbf{this}, x)\}_{\_}\{Val(\mathbf{this}, x)\}$
 { int tmp = this.Cell::get(); if(this.val2 != tmp) crash(); }
}
```

Figure 4. The TCell code

```
class DCell : Cell {
  define $Val_{\mathsf{DCell}}(x, v)$ as $false$
  define $DVal_{\mathsf{DCell}}(x, v)$ as $Val_{\mathsf{Cell}}(x, v)$

  public inherit int get()
  dynamic $\{DVal(\mathbf{this}, x)\}_{\_}\{DVal(\mathbf{this}, x) * ret = x\}$
    also $\{Val(\mathbf{this}, x)\}_{\_}\{Val(\mathbf{this}, x) * ret = x\}$

  DCell() {} dynamic $\{true\}_{\_}\{DVal_{\mathsf{DCell}}(\mathbf{this}, \_)\}$

  public override void set(int x)
  dynamic $\{DVal(\mathbf{this}, \_)\}_{\_}\{DVal(\mathbf{this}, \mathsf{x} * 2)\}$
    also $\{Val(\mathbf{this}, \_)\}_{\_}\{Val(\mathbf{this}, \mathsf{x})\}$
  { this.Cell::set(x * 2); }
}
```

Figure 5. The DCell class

The proof obligation (Behavioural Subtyping) is proved almost identically as for the get method.

$$\dfrac{\dfrac{\{Val(\mathbf{this}, v, \_)\}_{\_}\{Val(\mathbf{this}, x, v)\}}{\{\exists v.\ Val(\mathbf{this}, v, \_)\}_{\_}\{\exists v.\ Val(\mathbf{this}, x, v)\}}\ \text{VarElim}}{\{Val(\mathbf{this}, \_)\}_{\_}\{Val(\mathbf{this}, x)\}}\ \text{Conseq}$$

The last proof obligation (Dynamic dispatch), can be shown simply and is omitted, as is the verification of the undo method.

Interestingly, this class can inherit all three versions of swap: we only need to provide proofs for swap1 and swap2. For swap1 we must show

$$\dfrac{\dfrac{\{Val_{\mathsf{Cell}}(\mathbf{this}, v_1) * Val(\mathsf{c}, v_2)\}_{\_}\{Val_{\mathsf{Cell}}(\mathbf{this}, v_2) * Val(\mathsf{c}, v_1)\}}{\left\{\begin{array}{l}Val_{\mathsf{Cell}}(\mathbf{this}, v_1) * Val(\mathsf{c}, v_2)\\ * \mathbf{this}.\mathsf{bak} \mapsto \_ * \mathbf{this} : \mathsf{Recell}\end{array}\right\}_{\_}\left\{\begin{array}{l}Val_{\mathsf{Cell}}(\mathbf{this}, v_2) * Val(\mathsf{c}, v_1)\\ * \mathbf{this}.\mathsf{bak} \mapsto \_ * \mathbf{this} : \mathsf{Recell}\end{array}\right\}}}{\left\{\begin{array}{l}Val(\mathbf{this}, v_1) * Val(\mathsf{c}, v_2)\\ * \mathbf{this} : \mathsf{Recell}\end{array}\right\}_{\_}\left\{\begin{array}{l}Val(\mathbf{this}, v_2)\\ * Val(\mathsf{c}, v_1)\end{array}\right\}}$$

For swap2 we must satisfy (1), which can done trivially by defining $ValLeft$ for Recell as **this**.bak $\mapsto \_$.

5.2 Restriction: TCell

Now we consider a derived class, TCell, that restricts the behaviour of its base class. The code is given in Figure 4. It defines a field val2 that is expected to contain the same value as would be returned by calling get. Every time the set method is called, both representations are updated, hence the check method should never be able to call crash.

For this class, we must prove that it is valid to inherit the get method, (Inheritance):

$$\dfrac{\dfrac{\{Val_{\mathsf{Cell}}(x, v)\}_{\_}\{Val_{\mathsf{Cell}}(x, v) * ret{=}v\}}{\left\{\begin{array}{l}Val_{\mathsf{Cell}}(x, v)\\ * x.\mathsf{val2}{\mapsto}v\end{array}\right\}_{\_}\left\{\begin{array}{l}Val_{\mathsf{Cell}}(x, v)\\ * ret{=}v * x.\mathsf{val2}{\mapsto}v\end{array}\right\}}\ \text{Frame}}{\{Val_{\mathsf{TCell}}(x, v)\}_{\_}\{Val_{\mathsf{TCell}}(x, v) * ret{=}v\}}\ \text{Conseq}$$

We will omit the proof obligations for the set method. The check method requires that we prove (Body Verification), which follows.

```
$\{Val_{\mathsf{TCell}}(\mathbf{this}, \mathsf{x})\}$
$\{Val_{\mathsf{Cell}}(\mathbf{this}, \mathsf{x}) * \mathbf{this}.\mathsf{val2} \mapsto \mathsf{x}\}$
int tmp = this.Cell::get();
$\{Val_{\mathsf{Cell}}(\mathbf{this}, \mathsf{x}) * \mathbf{this}.\mathsf{val2} \mapsto \mathsf{x} * \mathsf{tmp} = \mathsf{x}\}$
if(this.val2 != tmp) {
    $\{Val_{\mathsf{Cell}}(\mathbf{this}, \mathsf{x}) * \mathbf{this}.\mathsf{val2} \mapsto \mathsf{x} * \mathsf{tmp} = \mathsf{x} * \mathsf{tmp} \neq \mathsf{x}\}$
      $\{false\}$crash();$\{false\}$
    $\{Val_{\mathsf{Cell}}(\mathbf{this}, \mathsf{x}) * \mathbf{this}.\mathsf{val2} \mapsto \mathsf{x} * \mathsf{tmp} = \mathsf{x}\}$
}
$\{Val_{\mathsf{Cell}}(\mathbf{this}, \mathsf{x}) * \mathbf{this}.\mathsf{val2} \mapsto \mathsf{x} * \mathsf{tmp} = \mathsf{x}\}$
$\{Val_{\mathsf{TCell}}(\mathbf{this}, \mathsf{x})\}$
```

Hence, the method can never call crash when its pre-condition is met.

Now, we consider inheriting the various swap methods. Firstly, swap can trivially be inherited. swap1 cannot be inherited:

$$\dfrac{\dfrac{\{Val_{\mathsf{Cell}}(\mathbf{this}, v_1) * Val(\mathsf{c}, v_2)\}_{\_}\{Val_{\mathsf{Cell}}(\mathbf{this}, v_2) * Val(\mathsf{c}, v_1)\}}{\left\{\begin{array}{l}Val_{\mathsf{Cell}}(\mathbf{this}, v_1) * Val(\mathsf{c}, v_2)\\ * \mathbf{this}.\mathsf{val2} \mapsto v_1 * \mathbf{this} : \mathsf{TCell}\end{array}\right\}_{\_}\left\{\begin{array}{l}Val_{\mathsf{Cell}}(\mathbf{this}, v_2) * Val(\mathsf{c}, v_1)\\ * \mathbf{this}.\mathsf{val2} \mapsto v_1 * \mathbf{this} : \mathsf{TCell}\end{array}\right\}}}{\left\{\begin{array}{l}Val(\mathbf{this}, v_1) * Val(\mathsf{c}, v_2)\\ * \mathbf{this} : \mathsf{TCell}\end{array}\right\}_{\_}\left\{\begin{array}{l}???\\ ???\end{array}\right\}}$$

Our proof fails because **this**.val2 $\mapsto v_1$, so we cannot establish the post-condition $Val(\mathbf{this}, v_2) * Val(\mathsf{c}, v_1)$ as this requires the field to have been updated to **this**.val2 $\mapsto v_2$. Hence, TCell would have to override the swap1 method.

In addition, this class does not satisfy (1) required by swap2. The state separate from Val_{Cell} depends on the value v, but $ValLeft$ does not take this as an argument. If this constraint was imposed on the system, then the TCell class would not be allowed.

5.3 Reuse: DCell

In this example we present a subtype of Cell that is not a well-behaved subtype in the traditional sense of behavioural subtyping. We define the class DCell in Figure 5; it is essentially a Cell that doubles the value it is set to. This breaks the standard substitutivity property, and hence is not allowed in other verification methodologies. However, as we shall see, it can be verified in our proof system.

We define the predicate family for Val_{DCell} to be $false$. This prevents clients calling a DCell using the Cell's interface. This reflects that fact that the use of inheritance in DCell is for code reuse. To inherit the get method we must show the (Inheritance) and (Behavioural Subtyping) proof obligations. The former can be proved as follows.

$$\dfrac{\dfrac{\{Val_{\mathsf{Cell}}(x, v)\}_{\_}\{Val_{\mathsf{Cell}}(x, v) * ret{=}v\}}{\{Val_{\mathsf{Cell}}(x, v) * x{:}\mathsf{DCell}\}_{\_}\{Val_{\mathsf{Cell}}(x, v) * ret{=}v * x{:}\mathsf{DCell}\}}}{\{DVal(x, v) * x{:}\mathsf{DCell}\}_{\_}\{DVal(x, v) * ret{=}v\}}$$

The latter proof obligation follows directly from the definition of **also** (Definition 1).

Consider, the following client code of the DCell class.

```
public void crash()
dynamic $\{false\}_{\_}\{false\}$
{ crash(); }

public void f(Cell c, DCell d)
dynamic $\{Val(\mathsf{c}, \_) * DVal(\mathsf{d}, \_)\}_{\_}\{Val(\mathsf{c}, 5) * DVal(\mathsf{d}, 10)\}$
{
```

```
class SubRecell : Recell {
 Stack ints;

 define $Val_{\mathsf{SubRecell}}(x, v_1, v_2, l)$ as
 $x.\mathsf{ints} \mapsto i * Stack(i, v_1 :: l) * ((l = [] \wedge v_2 = v_1) \vee l = v_2 :: \_)$

 SubRecell() dynamic $\{true\}\_\{Val(\mathbf{this}, \_, \_, \_)\}$
 { ints = new Stack(); ints.push(0); }

 public override int get()
 dynamic $\{Val(\mathbf{this}, v_1, v_2, l)\}\_\{Val(\mathbf{this}, v_1, v_2, l) * v_1 = ret\}$
 { return ints.readTop(); }

 public override void set(int x)
 dynamic $\{Val(\mathbf{this}, v_1, v_2, l)\}\_\{Val(\mathbf{this}, \mathsf{x}, v_1, v_1 :: l)\}$
 { ints.push(x); }

 public override void undo()
 dynamic $\{Val(\mathbf{this}, v_1, v_2, w2 :: l)\}\_\{Val(\mathbf{this}, v_2, \_, l) * v_2 = w2\}$
    also $\{Val(\mathbf{this}, v_1, v_2, [])\}\_\{Val(\mathbf{this}, v_2, \_, [])\}$
 { if(ints.length()>1) this.ints.pop(); }
}
```

Figure 6. Recell with unbounded backup

```
  if(c.set(5).get() != 5) crash();
  if(d.set(5).get() != 10) crash();
}
```

The specification for f amounts to showing that if the arguments c and d meet their specifications, then the method body never invokes the crash method. Ordinarily, this would be very hard to establish as DCell is not normally considered to be a behavioural subtype of Cell.

However, the power of the approach described in this paper is that it is possible (and quite simple) to show that the method f meets it specification. The verification proceeds directly from the dynamic specifications trivially. The first argument, c, can be any subtype of Cell that has the $Val(c, \_)$ predicate. Hence, we cannot call this with the first argument of type DCell.

Surprisingly, the DCell class can validly inherit the swap1 method, and satisfy the constraint (1) for swap2. The constraint is trivially satisfied by defining $ValLeft$ for DCell as $false$. Similarly, the swap1 method can be inherited trivially, as the specifications pre-condition is false for this class, so the method will never be called.

5.4 Altering internal representation: SubRecell

Finally we consider an example where we completely alter how data is represented in the base class. In Figure 6, we present a derived class of Recell called SubRecell that has an unbounded undo capacity. The code uses a Stack to store the values, and does not update the redundant fields it inherits from Recell; the code is clearer and simpler by not using the parent fields. As all the methods are overridden this class can be rather straightforwardly verified. We must simply show that each method satisfies the rule for behavioural subtyping (Behavioural Subtyping), and the method bodies implement the specification (Body Verification).

The method swap can trivially be inherited as it does not depend on the representation. However, swap1 cannot be inherited as it assumes the original fields are used. The constraint, (1), required for swap2 cannot be satisfied by this class.

5.5 Interplay between static and dynamic calls

Interestingly, the set method of Recell becomes considerably harder to verify if the call to get is turned into a dynamic call, rather than a direct/super call, that is

```
void set(int x)
dynamic $\{Val(\mathbf{this}, v, \_)\}\_\{Val(\mathbf{this}, \mathsf{x}, v)\}$
static $\{Val(\mathbf{this}, v, \_) * Val\_TO_{\mathsf{Recell}}(\mathbf{this})\}$ _ $\{Val(\mathbf{this}, \mathsf{x}, v)\}$
{ int tmp = this.get(); this.bak = tmp; this.Cell::set(x); }
```

The increased difficulty comes from considering all possible derived classes. If the derived class overrides the behaviour of get then this code could inflict untold damage! Hence, we must place a constraint on any method that inherits this code. This is done by adding Val_TO_{Recell} to the **static** pre-condition. We require that this predicate has the following properties

$$\begin{array}{l} Val\_TO_{\mathsf{Recell}}(\mathbf{this}) * Val(\mathbf{this}, x, v) \\ \quad \Rightarrow Val_{\mathsf{Recell}}(\mathbf{this}, x, v) * Val\_FROM_{\mathsf{Recell}}(\mathbf{this}) \end{array}$$

$$\begin{array}{l} Val_{\mathsf{Recell}}(\mathbf{this}, x, v) * Val\_FROM_{\mathsf{Recell}}(\mathbf{this}) \\ \quad \Rightarrow Val\_TO_{\mathsf{Recell}}(\mathbf{this}) * Val(\mathbf{this}, x, v) \end{array}$$

If the derived class alters the representation too much, then it is not possible to find solutions to these equations. Finding these solutions can be simplified[11] to proving that the following is a tautology.

$$\begin{array}{l} \forall xv.\ Val(\mathbf{this}, x, v) -\!\!* \\ \quad \begin{pmatrix} Val_{\mathsf{Recell}}(\mathbf{this}, x, v) * \\ \forall x'v'.\, Val_{\mathsf{Recell}}(\mathbf{this}, x', v') -\!\!* \ Val(\mathbf{this}, x', v') \end{pmatrix} \end{array}$$

The outer use of $-\!\!*$ is the TO predicate, and the inner one is the FROM predicate. With this additional predicate we can perform the verification, (Body Verification), as follows:

```
$\{Val(\mathbf{this}, v, \_) * Val\_TO_{\mathsf{Recell}}(\mathbf{this})\}$
  tmp = this.get();
$\{Val(\mathbf{this}, v, \_) * Val\_TO_{\mathsf{Recell}}(\mathbf{this}) * \mathsf{tmp} = v\}$
$\{Val_{\mathsf{Recell}}(\mathbf{this}, v, \_) * Val\_FROM_{\mathsf{Recell}}(\mathbf{this}) * \mathsf{tmp}{=}v\}$
$\{Val_{\mathsf{Cell}}(\mathbf{this}, v) * \mathbf{this}.\mathsf{bak}{\mapsto}\_ * Val\_FROM_{\mathsf{Recell}}(\mathbf{this}) * \mathsf{tmp}{=}v\}$
  this.bak = tmp;
$\{Val_{\mathsf{Cell}}(\mathbf{this}, v) * \mathbf{this}.\mathsf{bak}{\mapsto}v * Val\_FROM_{\mathsf{Recell}}(\mathbf{this})\}$
  this.Cell::set(x);
$\{Val_{\mathsf{Cell}}(\mathbf{this}, x) * \mathbf{this}.\mathsf{bak}{\mapsto}v * Val\_FROM_{\mathsf{Recell}}(\mathbf{this})\}$
$\{Val_{\mathsf{Recell}}(\mathbf{this}, x, v) * Val\_FROM_{\mathsf{Recell}}(\mathbf{this})\}$
$\{Val(\mathbf{this}, x, v)\}$
```

6. Conclusions and related work

In this paper we have considered the problem of verifying object-oriented programs that use inheritance in a number of different ways. We have defined a proof system that allows a derived class to (1) simply extend a base class (Recell); (2) restrict its base class's behaviour (TCell); (3) alter its base class's behaviour in a way incompatible with the standard view of behavioural subtyping (DCell); and (4) replace the representation of its base class (SubRecell). Even in the presence of these drastic changes, we are still able to inherit code without needing to see the actual implementation. As far as we are aware, no other modular proof system can verify all of these examples.

Poetzsch-Heffter and Müller (1999) present a logic with rules with both virtual (dynamic) and implementation (static) specifications. However, they do not explore the inter-relationship between the virtual and implementation specifications, and they do not consider how this distinction enables the inheritance of methods without reverification. The interaction with inheritance is the key to the examples presented in this paper.

The Java Modelling Language, JML (Leavens et al. 2006), also has similar notions to static (known as code contracts) and dynamic

[11] Some might say simplifying to uses of $-\!\!*$ is not simplifying at all!

specifications (known as non-code behaviour specifications). However the treatment of these specifications is different. Code contracts can be used to verify method invocations where the exact method can be statically determined. JML also takes a different approach to verifying overridden methods. It requires a method in a derived class to be verified against not only its specification, but also against all the specifications in its base classes. This has the advantage of simplifying the framework (i.e. by eliminating proof-theoretic notions such as refinement), but our work is motivated by wishing to avoid such repeated verification of overridden methods.

Dhara and Leavens (1996) propose a relaxation of the JML approach by defining slightly more liberal restrictions on the pre- and post-conditions of a method in a derived class that ensure behavioural subtyping.

JML does not currently define restrictions on inheriting methods. Addressing this, Müller (2002) and later (Müller et al. 2006) restrict invariants in a derived class to only mention fields/properties introduced in that class, and the class must preserve the invariants of its base classes. This means that methods can be inherited, although not all code satisfies the restrictions. For example, this approach can only deal with the Cell, and Recell examples presented in this paper.

Ruby and Leavens (2000) allow the invariant of a derived class to depend on fields from its base class. They provide a series of conditions for when it is valid to inherit a method into a class. Their work can deal with Cell, Recell and TCell examples. It still requires a derived class to satisfy the invariant of its base classes, so it cannot allow representation changes that are required for SubRecell and DCell.

Spec$^\sharp$/Boogie (Barnett et al. 2004) also allows the invariant of a derived class to restrict the invariant of its base class. This means it can deal with the Cell, Recell and TCell examples. Spec$^\sharp$/Boogie uses a single, "polymorphic" specification for each method that is interpreted in one of two ways, one for static/direct dispatch, and one for the dynamic dispatch. They do not explicitly separate the specification as we do in this paper, but their approach is clearly closely related. A polymorphic specification is one containing a distinguished symbol, typically written "1", that is replaced with the expression $type(\textbf{this})$ for dynamic dispatch, and with the expression C for the static dispatch, where C is the defining class.

This polymorphism can be justified quite succinctly using our notion of refinement, as follows (where we write $P[e]$ to mean the formula P where all occurrences of the symbol 1 are replaced with the expression e):

$$\{P[\mathsf{C}]\}\_\{Q[\mathsf{C}]\} \overset{\textbf{this}:\mathsf{C}}{\Longrightarrow} \{P[type(\textbf{this})]\}\_\{Q[type(\textbf{this})]\}.$$

Currently, Spec$^\sharp$/Boogie cannot deal with the SubRecell and DCell classes as it enforces that a derived class preserves the invariant of its base class.

In future work, we intend to pursue a more thorough comparison of our approach and the Spec$^\sharp$/Boogie system. In joint work with others (Parkinson et al. 2007) we propose the classic "gang of four" design patterns as a benchmark for the verification of object-oriented code. These patterns often make use of complicated aggregate structures. Class invariant-based approaches, such as Spec$^\sharp$, require significant extensions to handle these structures and their use (Leino and Schulte 2007). Early experiments suggest that our approach—using separation logic and abstract predicate families—requires no extensions to handle aggregate structures.

Note Independent to our work, Chin et al. (2008) suggest in these proceedings a similar approach to avoiding re-verification. Whilst working in the more traditional setting of class invariants, they propose a very similar use of static/dynamic method specifications in a separation logic. That two groups independently proposed the distinction between static and dynamic specifications is perhaps encouraging as to the naturalness of the basic idea.

Acknowledgments

We should like to thank Wei-Ngan Chin and Sophia Drossopoulou for discussions on this work, and Gary Leavens, Rustan Leino, Peter Müller, Clyde Ruby and Wolfram Schulte for discussions about related work.

References

M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

M. Barnett, K. R. M. Leino, and W. Schulte. The Spec$^\sharp$ programming system: An overview. In *Proceedings of CASSIS*, pages 49–69, 2005.

B. Biering, L. Birkedal, and N. Torp-Smith. Bi-hyperdoctrines, higher-order separation logic, and abstraction. *ACM TOPLAS*, 2007. To appear.

G. M. Bierman, M. J. Parkinson, and A. M. Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical Report 563, University of Cambridge Computer Laboratory, 2004.

W.-N. Chin, C. David, H. Nguyen, and S. Qin. Enhancing modular OO verification with separation logic. In *Proceedings of POPL*, 2008.

W. R. Cook, W. Hill, and P. Canning. Inheritance is not subtyping. In *Proceedings of POPL*, 1990.

K. K. Dhara and G. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of ICSE*, 1996.

C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proceedings of PLDI*, 1993.

M. Flatt, S. Krishnamurthi, and M. Felleisen. A programmer's reduction semantics for classes and mixins. Technical Report TR-97-293, Rice University, 1997. Corrected June, 1999.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.

S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *Proceedings of POPL*, pages 14–26, 2001.

N. Krishnaswami, J. Aldrich, and L. Birkedal. Modular verification of the subject-observer pattern via higher-order separation logic. In *Proceedings of FTfJP*, 2007.

G. T. Leavens and D. A. Naumann. Behavioral subtyping is equivalent to modular reasoning for object-oriented programs. Technical Report TR 06-36, Iowa State University, 2006.

G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.

K. R. M. Leino. Data groups: Specifying the modification of extended state. In *Proceedings of OOPSLA*, pages 144–153, 1998.

K. R. M. Leino and P. Müller. A verification methodology for model fields. In *Proceedings of ESOP*, 2006.

K. R. M. Leino and W. Schulte. Using history invariants to verify observers. In *Proceedings of ESOP*, 2007.

B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM TOPLAS*, 16(6):1811–1841, 1994.

P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer-Verlag, 2002. PhD thesis, FernUniversität Hagen.

P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62:253–286, 2006.

A. Nanevski, A. Ahmed, G. Morrisett, and L. Birkedal. Abstract predicates and mutable ADTs in Hoare Type Theory. In *Proceedings of ESOP*, 2007.

P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of CSL*, pages 1–19, 2001.

M. Parkinson, G. Bierman, J. Noble, and W. Schulte. Contracts for patterns. Unpublished note, 2007.

M. J. Parkinson. *Local Reasoning for Java*. PhD thesis, Computer Laboratory, University of Cambridge, 2005. UCAM-CL-TR-654.

M. J. Parkinson and G. M. Bierman. Separation logic and abstraction. In *Proceedings of POPL*, pages 247–258, 2005.

A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In *Proceedings of ESOP*, volume 1576 of *LNCS*, 1999.

J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of LICS*, pages 55–74, 2002.

C. Ruby and G. T. Leavens. Safely creating correct subclasses without seeing superclass code. *SIGPLAN Not.*, 35(10):208–228, 2000.

H. Yang. *Local reasoning for stateful programs*. PhD thesis, University of Illinois, July 2001.

A. Semantics of proof system

We give the semantics of our logic with respect to a state, σ, an interpretation of predicates, $\mathcal{I}$, and an interpretation of logical variables, $\mathcal{L}$. An interpretation of predicates maps predicate names to predicate definitions, where predicate definitions map a list of values to a set of states, that is:

$$\begin{array}{l}\mathcal{I} : \mathsf{Preds} \to (\mathsf{Vals}^* \to \mathcal{P}(\Sigma)) \\ \mathcal{L} : \mathsf{Vars} \to \mathsf{Vals}\end{array}$$

We define predicates in the standard way for a predicate calculus:

$$\sigma, \mathcal{I}, \mathcal{L} \models \alpha(\overline{X}) \iff \sigma \in (\mathcal{I}(\alpha)(\mathcal{L}(\overline{X})))$$

Definition 6. $\mathcal{I} \models \Delta$ *iff for all* σ *and* $\mathcal{L}$ *then* $\sigma, \mathcal{I}, \mathcal{L} \models \Delta$ *holds*

Given this definition, we can prove that any set of disjoint abstract predicate family definitions is satisfiable.

Lemma 7. *For any set of disjoint definitions,* $\mathsf{A}_1 \ldots \mathsf{A}_n$*, there exists an environment,* $\mathcal{I}$*, such that* $\mathcal{I} \models [\![\mathsf{A}_1]\!] \wedge \ldots \wedge [\![\mathsf{A}_n]\!]$*, provided that if* A_i *defines* α_{C} *and* A_j *defines* α_{C}*, then* $i = j$*.*

Next, we define the semantics of the judgements of the proof system. We use the standard semantics of a triple for separation logic, that is, if the pre-condition holds of the start state, then (1) the program will not fault (e.g. access unallocated memory); and (2) if the program terminates, then the final state will satisfy the post-condition.

Definition 8. $\mathcal{I} \models_n \{P\}\,\bar{\mathsf{s}}\,\{Q\}$ *iff* $(S, H), \mathcal{I}, \mathcal{L} \models P$ *then* $\forall m \leq n$*.*

- $S, H, \bar{\mathsf{s}} \longrightarrow^m$ *fault does not hold; and*
- *if* $S, H, \bar{\mathsf{s}} \longrightarrow^m S', H', \mathit{skip}$ *then* $(S', H'), \mathcal{I}, \mathcal{L} \models Q$*.*

Note that the step index n is used to deal with mutual recursion in method definitions.

We define $\mathcal{I} \models_n \Gamma$ to mean all the methods given in Γ meet their specifications for at least n steps.

Definition 9 (Semantics of method verification)**.**

$\mathcal{I}, \Gamma \models_{n+1} \mathsf{C.m}(\bar{\mathsf{x}})\colon \{P\}\_\{Q\}$ *iff*
$\quad\mathcal{I} \models_n \Gamma \Rightarrow \mathcal{I} \models_{n+1} \{P \wedge \mathbf{this} : \mathsf{C}\}\ \mathit{mbody}(\mathsf{C}, \mathsf{m})\ \{Q\}$

$\mathcal{I}, \Gamma \models_{n+1} \mathsf{C{::}m}(\bar{\mathsf{x}})\colon \{S\}\_\{T\}$ *iff*
$\quad\mathcal{I} \models_n \Gamma \Rightarrow \mathcal{I} \models_{n+1} \{S \wedge \mathbf{this} \neq \mathbf{null}\}\ \mathit{mbody}(\mathsf{C}, \mathsf{m})\ \{T\}$

$\mathcal{I} \models_0 \Gamma$ *always holds.*

$\mathcal{I} \models_{n+1} \Gamma$ *iff* $\forall \mathit{spec} \in \Gamma.\ \mathcal{I}; \Gamma \models_{n+1} \mathit{spec}$

We can now define the precise semantics of a judgement as follows.

Definition 10. $\Delta; \Gamma \models \{P\}\bar{s}\{Q\}$ *iff for all* $\mathcal{I}$ *and* n*, if* $\mathcal{I} \models \Delta$ *and* $\mathcal{I} \models_n \Gamma$*, then* $\mathcal{I} \models_{n+1} \{P\}\,\bar{\mathsf{s}}\,\{Q\}$*.*

That is, for all interpretations satisfying the predicate definitions Δ and assuming all the methods executed for at most n steps meet their specification given by Γ, then the statements $\bar{\mathsf{s}}$ meet their specification for at least $n+1$ steps.

The judgements for statement and method verification are sound with respect to the semantics.

Lemma 11.

1. *If* $\Delta; \Gamma \vdash \{P\}\bar{\mathsf{s}}\{Q\}$ *then* $\Delta; \Gamma \models \{P\}\,\bar{\mathsf{s}}\,\{Q\}$*.*
2. *If* $\Delta; \Gamma \vdash \mathsf{M}$ *in* E *then*
   $\forall \mathcal{I}.$ *if* $\mathcal{I} \models \Delta$ *then* $\forall n \forall \mathit{spec} \in \mathsf{M}.\ \mathcal{I}; \Gamma \models_n \mathit{spec}$

Our notion of refinement respects the weakening of assumptions and, hence, we can verify classes in a weaker context.

Lemma 12.

1. *If* $\Delta' \Rightarrow \Delta$ *and* $\Delta \vdash \{P_1\}\_\{Q_1\} \Longrightarrow \{P_2\}\_\{Q_2\}$*,*
   then $\Delta' \vdash \{P_1\}\_\{Q_1\} \Longrightarrow \{P_2\}\_\{Q_2\}$*.*
2. *If* $\Delta; \Gamma \vdash \mathsf{L}$ *and* $\Delta' \Rightarrow \Delta$*, then* $\Delta'; \Gamma \vdash \mathsf{L}$*.*

Finally, we state and outline the soundness proof for the program verification rule.

Theorem 13. *If a program and main body* $\bar{\mathsf{s}}$ *can be proved using the program verification rule, then* $\forall \mathcal{I}, n.\ \mathcal{I} \models_n \{\mathit{true}\}\,\bar{\mathsf{s}}\,\{\mathit{true}\}$*.*

Proof. Using Lemma 12.2, we can simplify the rule to the following.

$$\frac{\Delta; \Gamma \vdash \mathsf{L}_1 \qquad \cdots \qquad \Delta; \Gamma \vdash \mathsf{L}_n \qquad \Delta; \Gamma \vdash \{\mathit{true}\}\bar{\mathsf{s}}\{\mathit{true}\}}{\vdash \mathsf{L}_1 \ldots \mathsf{L}_n \bar{\mathsf{s}}}$$

where $\Gamma = \mathit{specs}(\mathsf{L}_1 \ldots \mathsf{L}_n)$ and $\Delta = \mathit{apf}(\mathsf{L}_1) \wedge \cdots \wedge \mathit{apf}(\mathsf{L}_n)$.

This rule assumes that Δ is satisfied, which we know by Lemma 7. The rest of the details are standard for the soundness of an object-oriented logic for partial correctness, for example see (Parkinson 2005). □