Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 212 (2008) 27–40
www.elsevier.com/locate/entcs

Separation Logic for Multiple Inheritance

Chenguang Luo[1]

Department of Computer Science
Durham University
Durham DH1 3LE, United Kingdom

Shengchao Qin[2]

Department of Computer Science
Durham University
Durham DH1 3LE, United Kingdom

Abstract

As an extension to Floyd-Hoare logic, separation logic has been used to facilitate reasoning about imperative programs manipulating shared mutable data structures. Recently, it has also been extended to support modular reasoning in Java-like object-oriented languages where only single inheritance is allowed. In this paper we propose an extension of separation logic to support also the reasoning for multiple inheritance in C++ -like languages. To cater for multiple inheritance, we modified the standard storage model for separation logic in a way that the correct reference to a field or a method can be easily determined. On top of this storage model, a set of proof rules are proposed. Our verification system also provides basic support for behavioral subtyping.

Keywords: Multiple Inheritance, Separation Logic, Verification

1 Introduction

Object-oriented programming languages offer a significant means to create software in a modular way. The mechanisms of inheritance and encapsulation allow program modules to provide sufficient information for others to make use of them, meanwhile hiding implementation details. This characteristic is excellent for code reuse and management, but it also calls for new formal approaches to reasoning about and verifying these OO programs.

Separation logic [14,3,15] has been proved to be an effective way to reason about pointer- and heap-related imperative programs. It brought forward a new concept

[1] Email: chenguang.luo@durham.ac.uk
[2] Email: shengchao.qin@durham.ac.uk

doi:10.1016/j.entcs.2008.04.051

in the runtime storage model, viz. the heap, and enhanced the expressiveness of Hoare logic over this model. The new logic operators it has introduced, separation conjunction and implication, not only specify explicitly that some part of the heap is disjoint from another part, but also allow local reasoning about pointers and heaps. This feature provides great support for modular reasoning, since each module's behavior can be held in the variables that it accesses (its so-called "footprint"). As a result, separation logic is an excellent tool for OO program verification.

Due to the fair properties of separation logic for OO verification, in this paper we propose an extension of separation logic to reason about programs written in a C++ -like object-oriented language with multiple inheritance. To achieve this objective, there are two main issues to deal with. The first is to extend the syntax and semantics of separation logic with method definitions and invocations, and (multiple) class inheritance, since the classical separation logic concentrates mostly on simple process-based languages. The second is to treat class inheritance and object polymorphism appropriately in the extension so as to cope with inherited classes and overridden methods. This paper's contribution mainly resides in the first aspect, that is, it introduces the syntactical constructs and related semantics of multiple inheritance into a modularized separation logic, and formalizes the assertions and verification rules accordingly. For the second aspect, we follow Liskov's behavioral subtyping ([4,7]) directly to make our work form a whole verification system for OO programs with multiple inheritance.

The remainder of this paper is organized as follows. Next section sets up the syntax of the C++ -like language used throughout this paper, and its semantics of class inheritance and method overriding in a syntactical way. Then the separation logic for this language is introduced, with the storage model expressing the program states, the assertion language and its semantics, and the verification rules. After that our approach of forcing behavioral subtyping is brought out, followed by an example to illustrate this model for multiple inheritance verification, and the last section summarizes the paper with related and future works.

2 An OO language with multiple inheritance

2.1 A core C++

Previous works on separation logic for object-oriented languages were mainly about languages supporting single inheritance, such as Java. This section's aim is to provide a kernel part of C++ which supports multiple inheritance of classes.

First we exhibit this core of C++ for verification. Any $\overline{s}$ means a (possibly

empty) sequence of s.

$$
\begin{array}{lll}
\mathit{prog} & := & \overline{\mathit{cdef};} \\
\mathit{cdef} & := & \texttt{class}\ C : \overline{C'}\{\overline{\mathit{fdef}}\ \overline{\mathit{mdef}}\} \\
\mathit{mdef} & := & [\texttt{virtual}]\ C\ *m(C_1\ *x_1, \ldots, C_n\ *x_n)\{\overline{s}\ \texttt{return}\ x;\} \\
\mathit{fdef} & := & C\ *f; \\
e & := & x \mid x{\rightarrow}\overline{C::}f \mid \texttt{null} \mid \texttt{this} \\
s & := & x{=}y{\rightarrow}\overline{C::}f;\ \mid x{=}(C*)y;\ \mid x{=}\texttt{new}\ C();\ \mid \texttt{delete}\ x;\ \mid x{\rightarrow}\overline{C::}f{=}e;\ \mid \\
 & & x{=}y{\rightarrow}\overline{C::}m(\overline{e});\ \mid C\ *x;\ \mid \{\overline{s}\}\ \mid ;\ \mid \texttt{if}\ (e == e)\ s\ \texttt{else}\ s
\end{array}
$$

This core language includes features such as class definitions, multiple inheritance, polymorphism and method overriding. Other features comprise field and variable definitions, the initialization and recycling of objects, variable assignment, and conditional constructs.

To keep the formalism simple, we leave out other C++ features that are of less interest. Since separation logic is mainly about the aliases and mutation of pointers to heap, all the variables in this core are assumed to be pointers. Arithmetic expressions and some other language constructs are also omitted. These features are to be added to our concern step by step in our future works towards practical implementation of this system.

Some of C++'s original semantics is also changed. For any field or variable x, as we do not focus on dangling pointers in this paper, it is assumed that $x = \texttt{null}$ if x is initially defined or later deleted. This aspect will be explored later.

At last are some conventions for the remainder of the paper. To traverse the class names, we use C and D with proper subscripts. For class fields and methods, f and m are used respectively. For general variables x (and sometimes y) is used. And for any tuple a, we denote $a.i$ as the i-th component of a.

2.2 Semantics for inheritance

Before the introduction of the semantics for program's execution, this section will bring about the semantics for multiple inheritance of our core C++ in a syntactical way, as a foundation of further runtime semantics.

As is in C++, the inheritance of classes is expressed in the class definition

$$\texttt{class}\ C : C_1, C_2, \ldots, C_n\{\overline{\mathit{fdef}}\ \overline{\mathit{mdef}}\}$$

carrying the meaning that C is a subclass of $C_1, C_2, \ldots, C_n$. In this case we denote $C \prec C_i$ and it is clear that they form a partial order among these classes. There is a synonym of this inheritance relationship that, assuming C has a fields list $f_1, \ldots, f_m$ defined in its body, and each C_i has a fields list $f_{i,1}, \ldots, f_{i,m_i}$, then C's complete fields list is $f_1, \ldots, f_m, C_1{::}f_{1,1}, \ldots, C_1{::}f_{1,m_1}, \ldots, C_n{::}f_{n,m_n}$. And this situation also applies to method inheritance.

This multiple inheritance, like C++, does not require all fields' and methods' names are distinct. Actually identical names of fields and methods are the very way

to promote dynamic polymorphism. However, it can also cause ambiguity when we refer to some fields of an object. For this issue we take the former inheritance `class` $C : C_1, \ldots, C_n$ and an object x typed C to describe the following cases:

- A field named f which is only defined in C_1. Then the reference $x{\rightarrow}f$ is equal to $x{\rightarrow}C_1{::}f$. And any other class indirectly inheriting C_1, without f defined, can refer to $x{\rightarrow}f$ equally as $x{\rightarrow}C_1{::}f$.
- A field named f which is defined only in both C and C_1. Then the references $x{\rightarrow}f$ and $x{\rightarrow}C_1{::}f$ might be distinct. This is similar as the case in single inheritance.
- A field named f which is defined only in both C_1 and C_2. Then the reference $x{\rightarrow}f$ is invalid, and the two f's that are overridden must be referred to with explicit specification $x{\rightarrow}C_1{::}f$ and $x{\rightarrow}C_2{::}f$. This also applies to other classes (indirectly) inheriting C. This case is unique in multiple inheritance and analogous as C++.
- A field named f which is defined only in C, C_1 and C_2. Then all the references $x{\rightarrow}f, x{\rightarrow}C_1{::}f$ and $x{\rightarrow}C_2{::}f$ are distinct. This is also the C++ case, unique in multiple inheritance.

These rules are applicable for methods overriding as well, except for two issues. First, both the method names and the parameter types are criteria to judge "identical names" of methods. Second, unlike fields, only virtual methods (defined with prefix "`virtual`") can be overridden.

Followed is the semantics of access to indirectly inherited fields and methods. Assume we have a chain of inheritance relationship $C \prec D_1 \prec D_2 \prec \cdots \prec D_n$, and a field f is defined in D_n. Then for an object x of C, f can be referred to with the expression $x{\rightarrow}D_1{::}D_2{::}\ldots{::}D_n{::}f$, where we modify the C++ syntax slightly to explicitly specify the inheritance chain instead of casting x.

With the semantics above, the rules to specify a unique field for $x{\rightarrow}f$ can be summarized as a function over the syntactical constructs of our language. Consider x as an object of class C and $\{D_n\}$ is the set of all superclasses of C. When we try to access the field $x{\rightarrow}f$, its syntactical meaning is as follows:

$$inh(x{\rightarrow}f) =_{df} \begin{cases} x{\rightarrow}\ D_{i_{m-1}}{::}D_{i_{m-2}}{::}\ldots{::}D_{i_k}{::}f, \text{ if there is a set } \mathcal{D} = \{D_{i_j} \mid \\ \quad D_{i_j} \prec D_{i_{j-1}}, j = 1 \ldots m, C = D_{i_m}\} \text{ and } f \text{ is in } D_{i_k} \\ \quad \text{and for any } D \in \{D_n\} \text{ that has a field } f, \text{ we have} \\ \quad D \in \mathcal{D} \text{ and } D_{i_k} \preceq D; \\ \bot, \quad \text{otherwise.} \end{cases}$$

$$inh(x{\rightarrow}\overline{D{::}}f) =_{df} x{\rightarrow}\overline{D{::}}f.$$

and these two mappings also apply to virtual method invocations. For non-virtual methods like $x{\rightarrow}\overline{D{::}}m$, we always have $inh(x{\rightarrow}\overline{D{::}}m) =_{df} x{\rightarrow}\overline{D{::}}m$.

An informal description of ambiguity is that there are two chains of inheritance with the bottom (subclass of all) C, and f occurs in classes of both chains. For other situations different from the one above, that is, f appears and only appears in one chain, then the f in the nearest class to C is the reference we need.

As is stated, dynamic polymorphism can be realized under this infrastructure.

Take a class inheritance `class` $C : D$ and a variable x typed D as example, x can also be initialized as an object of C via $x =$ `new` $C()$. In the situation that both C and D has the field f, the semantics of expression $x{\rightarrow}f$ is a reference to the f of $x{\rightarrow}C{::}f$ but not $x{\rightarrow}D{::}f$, and we must explicitly specify the latter if we want its reference. The verification system to be built later is based on such semantics.

Finally, in the remainder of this paper it is assumed that all the class inheritances are valid (no illegal circle in the partial order) and all the references are not ambiguous, since these should be guaranteed at compile-time.

3 Separation logic for multiple inheritance

3.1 The storage model

As is introduced, separation logic is about the reasoning of aliases and shared mutable objects. Because both such aliases and sharing occur on the heap, it is necessary to include the heap in our storage model, which is different from classical Hoare logic where variables are accessed based on the stack. Meanwhile stack is still needed in our model, since the heap can be modeled as a mapping from an address to a value, in which the address should be accessed via the stack. Our storage model, given as follows, is based on the standard model for separation logic.

$$\begin{aligned}
\mathsf{Stack} &=_{df} \mathsf{ProgVar} \rightharpoonup \mathsf{Loc} \\
\mathsf{Aux} &=_{df} \mathsf{AuxVar} \rightharpoonup \mathsf{Loc} \\
\mathsf{Heap} &=_{df} \mathsf{Loc} \rightharpoonup (\mathsf{ClassName} \times \mathsf{Field}) \\
\mathsf{Field} &=_{df} \{f \mid f : \mathsf{ClassName} \times \mathsf{FieldName} \rightharpoonup \mathsf{Loc}\}
\end{aligned}$$

Stack is a mapping from program variables to address locations Loc, which will not be defined further.

The auxiliary variables compose a significant part of the program state. Though they are not a real part of the storage, they can be useful as assistance to describe program states and specifications. Hence they are also involved in the storage model. These variables do not have fixed types and are supposed to capture any appropriate value from both the stack and the heap.

Note that our model of heap is slightly different from the classical separation logic, where heap is modeled as a mapping from locations (a subset of integers or values) to values, i.e. $\mathsf{Heap} =_{df} \mathsf{Loc} \rightharpoonup \mathsf{Value}^{+}$. The reason for us to diverge from this is that, as an object-oriented language, heap locations are not simply typed as integers or floats, but as more complicated objects of user-defined classes in order to support inheritance and method overriding. Hence the type information of an object must be recorded.

Meanwhile, similar as that a heap location can store many values in the classical separation logic, in our model an object can also have many fields with diverse types. As a result, beneath the heap there is one more set of mappings, Field, to record the information of an object's fields. Each element f of Field, denoting one object in the runtime environment, maps a pair of a field's (*type*, *name*) to its stored address, where the *name* is the name of the field, and the *type* is the class name

containing the field (but not the type of the field itself). This design is to facilitate multiple inheritance, since a class may inherit many fields with the same name from its ancestor classes, which form a lattice in lieu of a chain; hence both the name of the field and the actual class containing it are needed when accessing that field. The values mapped to by f are undefined for such pairs not contained in this object.

Besides the heap model itself, a union operation for two disjoint heaps is defined here to support the semantics later:

$$h_1 * h_2 =_{df} \begin{cases} h_1 \cup h_2, & \text{if } \mathrm{dom}(h_1) \cap \mathrm{dom}(h_2) = \emptyset; \\ \bot, & \text{otherwise.} \end{cases}$$

This operation conforms to the separation conjunction which will be brought in by the introduction of separation logic. Separation conjunction explicitly claims that two pointers point to disjoint part of heap and thus are not alias to each other, which reduces the intricacy of traditional types of logic for stating that a heap location is pointed to by a pointer exclusively.

Based on the aforesaid models, the definition of program state (also storage) model is a cartesian product of them.

$$\Sigma =_{df} \mathsf{Stack} \times \mathsf{Heap} \times \mathsf{Aux}$$

Later we will employ s, h and σ to go through instances of stacks, heaps and states.

3.2 Assertion language

As for separation logic, it adds two separation-related logical operators. One is separation conjunction $*$, and the other is the separation implication $-\!\!*$. In this paper only the first is utilized. Its semantics will be introduced later.

Next is the assertion language for our verification system, in which e and e' stand for variables, `null` and `this`, and f and f' denote field names:

$$\begin{aligned} P =_{df}\ & \mathsf{true} \mid \mathsf{false} \mid \neg P \mid P \wedge P \mid P \vee P \mid P \Rightarrow P \mid P * P \mid \\ & e = e' \mid e \rightarrow \overline{C} :: f = e' \rightarrow \overline{C} :: f' \mid e \rightarrow \overline{C} :: f \mapsto e' \mid e : C \mid \exists X.P \mid \forall X.P \end{aligned}$$

It is worth noting here that we do not have the assertion emp in standard separation logic to express empty heap. Instead, the assertion $x =$ `null` is used to express that no part of the heap is allocated for the variable x and thus it points to a reserved area `null`. This assertion not only depicts empty heap but also specifies the variable owning no corresponding allocated heap, which is more accurate without the presence of dangling pointers.

With the storage model as a foundation, each assertion is interpreted as a set of

states.

$$
\begin{array}{ll}
[\![\mathsf{true}]\!] & =_{df}\ \Sigma \\
[\![\mathsf{false}]\!] & =_{df}\ \emptyset \\
[\![\neg P]\!] & =_{df}\ \Sigma \setminus [\![P]\!] \\
[\![P \wedge Q]\!] & =_{df}\ [\![P]\!] \cap [\![Q]\!] \\
[\![P \vee Q]\!] & =_{df}\ [\![P]\!] \cup [\![Q]\!] \\
[\![P \Rightarrow Q]\!] & =_{df}\ [\![\neg P \vee Q]\!] \\
[\![P * Q]\!] & =_{df}\ \{(\sigma.1, \sigma.2, \sigma.3) \mid \sigma \in \Sigma \wedge \exists h_1, h_2\ .\ (\sigma.2 = h_1 * h_2 \\
 & \qquad \wedge (\sigma.1, h_1, \sigma.3) \in [\![P]\!] \wedge (\sigma.1, h_2, \sigma.3) \in [\![Q]\!])\} \\
[\![e = e']\!] & =_{df}\ \{\sigma \mid \sigma.1(e) = \sigma.1(e')\} \\
[\![e \rightarrow \overline{C::} f \mapsto e']\!] & =_{df}\ \{\sigma \mid \sigma.2(\sigma.1(e)).2(C_1, f) = \sigma.1(e')\} \\
 & \quad \text{where } C_1 \text{ is the nearest class' name to } f \text{ in } inh(e \rightarrow \overline{C::} f) \\
[\![e \rightarrow \overline{C::} f = e' \rightarrow \overline{C'::} f']\!] & =_{df}\ \{\sigma \mid \sigma.2(\sigma.1(e)).2(C_1, f) = \sigma.2(\sigma.1(e')).2(C_1', f')\} \\
 & \quad \text{where } C_1 \text{ and } C_1' \text{ are the nearest classes' names to } f \text{ and} \\
 & \quad f' \text{ in } inh(e \rightarrow \overline{C::} f) \text{ and } inh(e' \rightarrow \overline{C'::} f'), \text{ respectively} \\
[\![e : C]\!] & =_{df}\ \{\sigma \mid \sigma.2(\sigma.1(e)).1 = C\}
\end{array}
$$

For the semantics of these assertions, the ones in ordinary logic are easy to understand. The separation conjunction $*$ means that, if both P and Q hold for a state σ, and the heap locations referred to by P are different from those by Q, then $P*Q$ also holds for σ, with the additional semantics that their orbits over the heap are disjoint.

The $\mapsto$ considers the field $\overline{C::}f$ of e as a pointer pointing to (or mapped to by the heap) the semantics of e'. This assertion allows us to reason about program variables' fields, where the fields are interpreted with the predefined function *inh* for inheritance purposes.

The $e : C$ is a test of a variable (or rather a location)'s real type in the program execution time. This is dynamic due to polymorphism. The result is possibly the variable's defined type or any of its subtype.

At last, the $\exists X.P$ and $\forall X.P$ are interpreted as in normal first-order logic, where X is an auxiliary variable. And for $\exists X.P(X)$ we use $P(\text{-})$ for short.

3.3 Rules

In this section we present the rules exploited in our verification system. These rules are in the following triple form:

$$\Gamma \vdash \{P\}\ \overline{s}\ \{Q\}$$

where $\Gamma := \epsilon \mid \{P\}\ C.m(\overline{x})\ \{Q\}, \Gamma$ is the statical result of verification, consisting of verified method specifications. The triples are common Hoare logic triples written in the former assertion language and our kernel C++'s syntax.

Of all the rules in our system, first are the structural ones.

$$\frac{P \Rightarrow P' \quad \Gamma \vdash \{P'\}\ \overline{s}\ \{Q'\} \quad Q' \Rightarrow Q}{\Gamma \vdash \{P\}\ \overline{s}\ \{Q\}}\ (conseq) \qquad \frac{\Gamma \vdash \{P\}\ \overline{s}\ \{Q\}}{\Gamma \vdash \{\exists X.P\}\ \overline{s}\ \{\exists X.Q\}}\ (var\text{-}elim)$$

$$\frac{\Gamma \vdash \{P\}\ \overline{s}\ \{Q\}}{\Gamma \vdash \{P * R\}\ \overline{s}\ \{Q * R\}}\ (frame)$$

where no variable in $\mathrm{mods}(\overline{s})$ appears freely in R.

The frame rule is a substitute of the "rule of constancy" in classical Hoare logic. By using this rule, the reasoning of local heap can be extended to a global one. For instance, if a recursive method call always uses a separate area of heap at a new time of invocation, then its behavior can be modeled locally to this method body first, and then extended to the heap containing other instances of invocation of this method using this rule.

Next are the canonical rules in ordinary Hoare logic. These rules include skip, instruction sequence, blocks and the if conditions.

$$\Gamma \vdash \{P\}\ ;\ \{P\}\ (skip)$$

$$\frac{\Gamma \vdash \{P\}\ s_1\ \{R\} \quad \Gamma \vdash \{R\}\ s_2\ \{Q\}}{\Gamma \vdash \{P\}\ s_1; s_2\ \{Q\}}\ (seq) \qquad \frac{\Gamma \vdash \{P\}\ \overline{s}\ \{Q\}}{\Gamma \vdash \{P\}\ \{\overline{s}\}\ \{Q\}}\ (block)$$

$$\frac{\Gamma \vdash \{P \wedge (e_1{=}e_2 \vee e_1{\mapsto}e_2)\}\ s_1\ \{Q\} \qquad \Gamma \vdash \{P \wedge \neg e_1{=}e_2 \wedge \neg e_1{\mapsto}e_2\}\ s_2\ \{Q\}}{\Gamma \vdash \{P\}\ \texttt{if}\ (e_1 == e_2)\ s_1\ \texttt{else}\ s_2\ \{Q\}}\ (if)$$

For assignment, there are two cases to be considered in our syntax. One is from object field to another variable, and the other is the opposite direction.

$$\Gamma \vdash \{y{\rightarrow}\overline{C{::}}f \mapsto X\}\ x{=}y{\rightarrow}\overline{C{::}}f;\ \{x = X\}\ (assign\text{-}1)$$
$$\Gamma \vdash \{e = X\}\ x{\rightarrow}\overline{C{::}}f{=}e;\ \{x{\rightarrow}\overline{C{::}}f \mapsto X\}\ (assign\text{-}2)$$

The two rules of assignment take use of auxiliary variable X instead of the traditional assignment rule using substitution in assertions. The auxiliary variable introduced can be eliminated later in a proof. Note that an assignment also changes the heap location that a variable points to.

Next is a special case of assignment: the downcast.

$$\Gamma \vdash \{(y{=}X \wedge y{:}C) \vee y{=}\texttt{null}\}\ x{=}(C\ *)y;\ \{x = X\}\ (downcast)$$

As downcasting requires that a variable of a base class be casted down to an object of its subclass, it is essential to check whether the variable is pointing to an object of the subclass; if not, the downcasting cannot be performed.

The rules below are for fields manipulation.

$$\Gamma \vdash \{\texttt{true}\}\ C\ *x;\ \{x{=}\texttt{null}\}\ (def)$$

$$\Gamma \vdash \{x{=}X\}\ x{=}\texttt{new}\ C();\ \{x{=}Y \wedge \neg X{=}Y \wedge x{:}C\}\ (new)$$

where Y is a fresh auxiliary variable.

$$\Gamma \vdash \{x \mapsto \text{-}\}\ \texttt{delete}\ x;\ \{x{=}\texttt{null}\}\ (\mathit{delete})$$

For variable definition, it is supposed that the compiler will prevent re-definition statically, and hence the pre-condition is not needed. The operations of `new` and `delete` have opposite behaviors on the heap, as is expressed in the rules, one allocates a fresh location in the heap, while the other disposes some part of the heap.

And the last are the rules for methods definition and invocation.

$$\frac{\Gamma \vdash \{P \wedge \texttt{this}{:}C\}\ \overline{C\ *\ y};\overline{y}{=}\overline{x};\overline{s}[\overline{y}/\overline{x}]\ \{Q[x/\texttt{ret}]\} \quad m(\overline{C'\ *\ x'})\{\overline{s}\ \texttt{return}\ x;\}}{\Gamma \leftarrow \{P\}\ C.m(\overline{x})\ \{Q\}, \Gamma}\ (\mathit{m\text{-}def})$$

where `ret` is the return value of the method and
$\overline{y}$ does not occur freely in the method body.

This rule first ensures that this method is defined in the right class C. After that the method body is checked to agree with the specification $\{P\}\ \overline{s}\ \{Q[x/\texttt{ret}]\}$. Then the specification is added to the statical environment to support potential method invocation, which is depicted in the following rule.

$$\Gamma_1, \{P\}\ D.m(\overline{x})\ \{Q\}, \Gamma_2 \vdash \{\theta(P) \wedge (\neg y{=}\texttt{null})\}\ x{=}y{\rightarrow}\overline{C}{::}m(\overline{e});\ \{\theta(Q)\}\ (\mathit{m\text{-}call})$$

where $\theta = [\overline{e}, y, x\ /\ \overline{x}, \texttt{this}, \texttt{ret}]$ and D is the last class name before m in $\mathit{inh}(y{\rightarrow}\overline{C}{::}m)$.

The above rule verifies the call of a overridden method from an object y. It has the inverse behavior of the last rule by substituting the actual parameters with the formal ones to fit in the specification recorded in Γ. When using this rule the static type of y should be judged.

In this section we have combined separation logic with a OO language with multiple inheritance. This logic has a semantics based on multiple inheritance and heap model of storage, and has a series of formal verification rules upon this model. However, when using such rules to derive a program's proof we still have to consider the polymorphism and the specification compatibility in method overriding, which will be the main topic of next section.

4 Behavioral subtyping

As our verification is performed over an object-oriented language with multiple inheritance, it is necessary to deal with dynamic dispatching during program execution time, for the possible inheritance and the resulted method overriding.

For this issue we turn to Liskov's behavioral subtyping ([4,7]). Since upcasting could occur in any place of a program, it must be ensured that any object of a subclass should behave like an object of its superclass. That is, for any of this object's method overriding the superclass' counterpart, its specification should be consistent with the superclass' method's. Thus no matter an object belonging to

a class or its subclass appears, it will conform to the expected specification. And this is formalized with *behavioral subtypes*, say, a type C is a behavioral subtype of D, if for each virtual method m of C overriding that of D, with the specifications $\{P_C\}\_\{Q_C\}$ and $\{P_D\}\_\{Q_D\}$, the implications $P_D \Rightarrow P_C$ and $Q_C \Rightarrow Q_D$ hold.

In the view of canonical Hoare logic, this definition of behavioral subtyping is another description of the rule of consequence. If all the classes of a program conform to such restrictions, then in verification it is safe to ignore the real type of an object during execution time, since the behavior of the object's methods will always comply with the specifications, no matter those for the base class or for its inherited classes.

Parkinson and Biermann ([12]) proposed a concept of *compatible specifications*, as a generality of behavioral subtyping: $\forall \overline{s}$, if we have $\{P_D\}\ \overline{s}\ \{Q_D\} \Rightarrow \{P_C\}\ \overline{s}\ \{Q_C\}$, namely, a proof of the latter with the only premise of the former, then the latter is defined as *compatible* with the former. The aim of behavioral subtyping is to ensure that each overriding method $C.m(\overline{x})$ has a specification compatible with its counterpart in the superclass, $D.m(\overline{x})$.

To achieve this goal, a possible way is to make a "conjunction" of the base class' method's specification and the subclass' method's. Such specification could carry the conditions specified for both classes' method and be used appropriately according to the real type of objects.

Consider two specifications for a virtual method m in class C and its base class D. m in C overrides m in D, and their specifications are as follows, respectively:

$$\{P_D\}\ \_\ \{Q_D\} \qquad \{P_C\}\ \_\ \{Q_C\}$$

If we can make a specification consistent with both two above, then whenever the method m of an object x, with static type of class D, is invoked, this specification can be used even if $x : C$. This can be accomplished by the following means:

$$\left\{ \begin{array}{l} (P_C \wedge I_C) \vee \\ (P_D \wedge I_D) \end{array} \right\} \ \_\ \left\{ \begin{array}{l} (Q_C \wedge I_C) \vee \\ (Q_D \wedge I_D) \end{array} \right\}$$

where assertions P_C, Q_C, P_D and Q_D are the same as above, while I_C and I_D are invariants of type identity. It can be shown that this is compatible with both m's in C and D. To illustrate, for $\{P_D\}\ \_\ \{Q_D\}$ as a specification of $D.m(\overline{x})$, we can set the invariant as $\texttt{this} : D$ and modify the specification as $\{P_D \wedge \texttt{this}{:}D\}\ D.m(\overline{x})\ \{Q_D \wedge \texttt{this}{:}D\}$, which can be derived just from our specification conjunction. Also an auxiliary variable may be set to specify the different cases, which can be eliminated with the auxiliary variable elimination rule later.

In our case of multiple inheritance, suppose that there are several chains of class inheritance relationship. Then for each of such chain $C_n \prec C_{n-1} \prec \cdots \prec C_1$ and each overridden method m of these classes, a conjunction of different specifications of diverse implementations of m should be made available. Without loss of generality, assume m is overridden in each C_i with specification $\{P_i\}\ C_i.m\ \{Q_i\}$, and then the total specification to capture all behaviors of different m's in this inheritance chain

should be

$$\{\bigvee_{i=1}^{n}(P_i \wedge I_i)\}\ m\ \{\bigvee_{i=1}^{n}(Q_i \wedge I_i)\}$$

which can be used at any call site of m belonging to an object with static type of any C_i.

5 An example

This section depicts an example as an illustration of our former means. It is based on the task-display example from Stroustrup [16,17] and has some modifications to cater for our language.

Consider a simulation of a network of computers. There are mainly two types of beings: tasks and displays. Tasks provide basic facilities for co-routine style behaviors of the network, and displays provide means for the monitoring of object states in the network. For these two concepts, two classes are defined as `Task` and `Display`. `Task`'s objects include network nodes called switches, and `Display` contains elements like communication lines in the network. These two are naturally modeled as `Switch` and `Line`, subclasses of the former two base ones.

Ideally `Task` and `Display` are disjoint because they share no similarity. However, a computer terminal, whose class is denoted `Terminal`, can act both as a task node and a monitor device. Thus the best way is to make `Terminal` inherit both `Task` and `Display`. In the following example we omit the definition of `Switch`, `Line` and other indifferent methods in our view for the space limit. For more detailed background of these examples, the bibliography is recommended.

Below are the codes for our examples. The class `Node` is added for support.

```
class Node { ... };
class Task {
  Node *n;
  virtual Node *trace
   (Node *an) {
    this->n = an;
    return this->n;
  }
};
```

```
class Display {
  Node *n;
  virtual Node *trace(Node *an) {
    Node *tmp;
    if (an == this->n) tmp = an;
    else tmp = this->n;
    return tmp;
  }
};
```

```
class Terminal : Display, Task {
  Node *trace(Node *an) {
    Node *tmp;
    if (this->Display::n == this->Task::n)
      tmp = this->Display::trace(this->Task::n);
    else tmp = this->Task::trace(an);
    return tmp;
  }
};
```

The results of the three methods' verification is depicted as a Γ built up by their specifications:

$$\{(this{\rightarrow}n \mapsto N_1 \wedge an{=}N_1) \vee (this{\rightarrow}n \mapsto N_1 \wedge an = N_2 \wedge \neg N_1{=}N_2)\}$$
$$\texttt{Display.trace(Node} * an)$$
$$\{this{\rightarrow}n \mapsto N_1 \wedge N_1{=}an \wedge an{=}\texttt{ret}\}$$

$$\{an = N_1\}\ \texttt{Task.trace(Node} * an)\ \{this{\rightarrow}n \mapsto N_1 \wedge N_1{=}an \wedge an{=}\texttt{ret}\}$$

$$\left\{\begin{array}{l}(this{\rightarrow}\texttt{Display::}n \mapsto N_1 * this{\rightarrow}\texttt{Task::}n \mapsto N_1) \vee \\ \quad ((this{\rightarrow}\texttt{Display::}n \mapsto N_1 * this{\rightarrow}\texttt{Task::}n \mapsto N_2) \wedge \neg N_1{=}N_2)\end{array}\right\}$$
$$\texttt{Terminal.trace(Node} * an)$$
$$\left\{\begin{array}{l}((this{\rightarrow}\texttt{Display::}n \mapsto N_1 * this{\rightarrow}\texttt{Task::}n \mapsto N_1) \Rightarrow \\ \qquad ((this{\rightarrow}\texttt{Display::}n \mapsto N_1 * this{\rightarrow}\texttt{Task::}n \mapsto N_1) \wedge \texttt{ret}{=}N_1)) \wedge \\ (((this{\rightarrow}\texttt{Display::}n \mapsto N_1 * this{\rightarrow}\texttt{Task::}n \mapsto N_2) \wedge \neg N_1{=}N_2) \Rightarrow \\ (this{\rightarrow}\texttt{Display::}n \mapsto N_1 * this{\rightarrow}\texttt{Task::}n \mapsto N_3 \wedge an{=}N_3 \wedge an{=}\texttt{ret}))\end{array}\right\}$$

To deal with behavioral subtyping, as is stated in the paper, a "conjunction" of the three methods' specifications can be made with an invariant to distinguish among different dynamic situations. However, in our example we only have to conjugate the specifications of `Display.trace`'s and `Terminal.trace`'s, and also `Task.trace`'s and `Terminal.trace`'s, separately, since these two are different chains of inheritance.

6 Conclusions

This paper brings out an extension of the separation logic for the reasoning of object-oriented programs with multiple inheritance. It has condensed the syntax of C++ to a core and set up a storage model for it based on the canonical separation logic. The verification rules are proposed according to this model, and behavioral subtyping is considered for the sake of runtime type decision. Finally an example is utilized to illustrate this whole approach.

6.1 Related work

Separation logic [14,3,15] is a significant extension of Hoare logic to boost reasoning about programs with pointers and heap manipulation. It allows reasoning to be done locally for a single part of heap without interfering with others, and also allows the result of reasoning to be extended again to global environment. Therefore many works took use of this modularity. O'Hearn et al. [10] used the separation conjunction and the hypothetical frame rule to separate one module's internal resources from those accessed by its clients which was a first step for separation logic towards modular program verification. O'Hearn [9] also employed the analogous approach in concurrent program verification to describe resources in critical region of concurrent processes. Due to this modularity, separation logic is highly suitable

for the verification of object-oriented programs. Another former attempt to introduce separation logic into the field of OO verification belongs to Parkinson [11]. In that work he found good means (abstract predicate family) to depict methods specifications and to rebuild the logic for Java program reasoning. We intend to follow his route to construct another extension of separation logic for a more general case of class inheritance. In our case the chains of inheritance might have intersections, which are different from single inheritance in both syntax and semantics.

To support reasoning about the type hierarchy and inheritance of object-oriented programming languages, and to capture their important properties such as polymorphism and diverse implementations, the concept of behavioral subtyping has been developed, as in Liskov [4], Liskov and Wing [7], Meyer [8], Dhara and Leavens [2], Leavens and Naumann [5], Parkinson [11], Parkinson and Biermann [13], and Chin et al. [1]. The first ([4]) was one of the beginners to develop the relationship between data abstraction and type hierarchy. Later Liskov and Wing proposed the behavioral subtyping as a general approach to dealing with inheritance and polymorphism ([7]). Another method to enforce that the new specification should also inherit the old one in class inheritance was introduced by Meyer ([8]) for Eiffel. Those two techniques were combined by Dhara and Leavens ([2]), and were eventually proven to be equivalent by Leavens and Naumann ([5]). The rest were practices, using behavioral subtyping or not, to process class hierarchy and inheritance in modeling and verification of object-oriented languages. Chin et al. [1] divided the method specification into finer-grained static and dynamic ones to prevent repeated verification and meanwhile to preserve behavioral subtyping. Parkinson and Biermann ([13]) used their abstract predicate family to cope with the same problem and could even handle cases there were beyond the capability of behavioral subtyping. In our work we took the compatible specification to guarantee behavioral subtyping. However, as a matter of fact, the way to treat with type hierarchy and class inheritance is orthogonal to our work. Therefore any of the aforesaid approach may be utilized.

6.2 Future work

Beyond this work, there are still several possible directions calling for further research. One of them aims at the defect of this work, viz. the re-verification of the inherited virtual methods. Since these methods may have different implementations from their antecedents, it may be necessary to treat them as totally new methods for re-verification, which requires much extra workload. Some feasible solutions for this include the works of Chin et al. ([1]), which separated method specifications to static and dynamic ones, where only the latter needed re-verification, and Parkinson and Biermann ([13]) who invented abstract predicate families to reduce the necessity of re-verification.

Another important future work is to give an implementation of this extension of separation logic. Before this more features of C++ must be added to the language, such as stack variables for basic data types, arithmetic operations and access control. Then the model and the rules should be combined to some (semi-) automatic reasoners for implementation.

Acknowledgement

Chenguang Luo is funded by Doctoral Fellowship Scheme of Durham University. Shengchao Qin is supported in part by the EPSRC project EP/E021948/1.

References

[1] W. Chin, C. David, H. H. Nguyen, and S. Qin. Enhancing modular OO verification with separation logic. San Francisco, USA, January 2008. The 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08).

[2] K. Dhara and G. Leavens. Forcing behavioral subtyping through specification inheritance. Berlin, Germany, March 1996. Proceedings of the 18th International Conference on Software Engineering (ICSE-18).

[3] S. Isthiaq and P. O'Hearn. BI as an assertion language for mutable data structures. London, United Kingdom, January 2001. The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'01).

[4] B. Liskov. Data abstraction and hierarchy. ACM SIGPLAN Notices, 23(5):1734, May 1988.

[5] G. Leavens and D. Naumann. Behavioral subtyping is equivalent to modular reasoning for object-oriented programs. Technical Report 06-36, Department of Computer Science, Iowa State University, 2006.

[6] J. Loeckx, K. Sieber, and R. Stansifer. *The foundations of program verification*. John Wiley and Sons, 1984.

[7] B. Liskov and J. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811-1841. November 1994.

[8] B. Meyer. Eiffel: the language. *Object-Oriented Series*. Prentice Hall. New York, 1992.

[9] P. O'Hearn. Resources, concurrency, and local reasoning. Proceedings of CONCUR'04, LNCS 3170, pp49-67.

[10] P. O'Hearn, H. Yang, and J. Reynolds. Separation and information hiding. Venice, Italy, January 2004. The 31th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'04).

[11] M. Parkinson. *Local reasoning for Java*. PhD thesis, Cambridge University, August 2005.

[12] M. Parkinson and G. Biermann. Separation logic and abstraction. Long Beach, California, USA, January 2005. The 32th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'05).

[13] M. Parkinson and G. Biermann. Separation logic, abstraction and inheritance. San Francisco, California, USA, January 2008. The 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08).

[14] J. Reynolds. Intuitionistic reasoning about shared mutable data structure. Millennial Perspectives in Computer Science, Proceedings of the 1999 Oxford–Microsoft Symposium in Honour of Sir Tony Hoare, 1999.

[15] J. Reynolds. Separation logic: A logic for shared mutable data structures. In *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[16] B. Stroustrup. *The C++ programming language*. Addison-Wesley, 1991.

[17] B. Stroustrup. Multiple inheritance for C++. *The C/C++ Users Journal*, May 1999.

[18] Daniel Wasserrab, Tobias Nipkow, Gregor Snelting, and Frank Tip. An operational semantics and type safety proof for multiple inheritance in C++. In OOPSLA'06: Object-Oriented Programming, Systems, Languages, and Applications. ACM Press, 2006.