A Modular Reasoning System Using Uninterpreted Predicates for Code Reuse

Crystal Chang Din, Einar Broch Johnsen, Olaf Owe, and Ingrid Chieh Yu

Department of Informatics, University of Oslo
P.O. Box 1080 Blindern, N-0316 Oslo, Norway
Email: {crystald,einarj,olaf,ingridcy}@ifi.uio.no

Abstract

This paper proposes a modular proof system based on uninterpreted predicates. The proposed proof system allows modular reasoning about programs with an open-world assumption, which goes beyond behavioral subtyping. The proof system enables modular reasoning about languages with very flexible code reuse mechanisms, such as traits and deltas in the context of object-oriented programming. Whereas related work on incremental proof systems prove soundness in terms of internal consistency, this paper establishes both soundness and relative completeness of the proposed proof system by relating it to a standard proof system for a simple object-oriented language. The applicability of the approach is demonstrated on different code reuse mechanisms: unrestricted class inheritance, delta-oriented programming, and trait-based programming.

Keywords: Modular programming, Modular reasoning, Code reuse, Uninterpreted predicates, Program verification, Early reasoning, Delayed specification, Soundness, Completeness

1. Introduction

Modern society is increasingly based on software which is constantly revised and updated. Code reuse and modular software development simplify the maintenance of complex software and shorten the development process until a new or enhanced release of a software program can be released. To support a development process based on flexible reuse and adaptation of software modules, it is crucial to be able to statically reason about the behavior of each module before composing reused modules into programs.

Class hierarchies in object-oriented programming provide a structured means to study software evolution in terms of reuse and adaptation of software modules. In object-oriented programs, class hierarchies typically grow as subclasses

[☆]This work was done in the context of the projects *CUMULUS: Semantics-based Analysis for Cloud-Aware Computing* and *IoTSec: Security of Internet of Things.*

are gradually developed. The code can be refactored; one method may move between a subclass and a superclass, another method may get reprogrammed with slightly different behavior, and a third method may get deleted. Software developers use code composition and reuse mechanisms to better structure this evolution. In particular, code reuse mechanisms enable existing modules to be adapted to meet new and unanticipated needs.

Class inheritance in object-oriented programming is a well-known structuring mechanism to support code reuse with adaptation: A class may extend its superclasses with new methods, possibly *overriding* existing ones. This flexibility significantly complicates reasoning about the behavior of methods as the method activation at runtime depends on the actual class of the called object. The behavior of a method may change indirectly due to calls to other, redefined methods. Modern code reuse and adaptation mechanisms, such as *traits* [14, 30] and *deltas* [27, 29], enable more radical changes to the code units by adding or removing methods and fields or by wrapping methods in the modified classes. In contrast to class inheritance, where the subclass is developed to extend one particular superclass, a trait or delta may be applied in several, different contexts.

Software verification systems impose *restrictions* in order to control code reuse and evolution. One approach is to rely on a *closed world assumption*; i.e., the proof rules assume that the final, complete code is available at reasoning time (e.g., [25]). This approach does not impose restrictions on software reuse, but severely reduces the applicability of the proof system; for example, libraries are often designed to be extended. Moreover, the closed world assumption contradicts inheritance as an object-oriented design principle, intended to support incremental development and analysis. If the reasoning system relies on the world being closed, extending the class hierarchy requires a costly reverification of previously analyzed code.

An alternative approach is to reflect an *open world assumption* in the verification system, but to constrain how methods may be redefined. To avoid reverification, any redefinition of a method must *preserve* the behavior of the method being redefined and in particular the method's contract; i.e., the pre- and postconditions for its body must remain valid. Best known as *behavioral subtyping* (e.g., [2, 3, 19, 21, 22, 26]), this approach achieves incremental reasoning by limiting the possibilities for method overriding, and thereby code reuse. Once a specification has been given for a method, this specification must be respected by later redefinitions. Proof systems with behavioral subtyping have also been proposed for delta-oriented programs [17]. However, behavioral subtyping has been criticized for being overly restrictive and is often violated in practice [31]. A main difficulty with behavioral subtyping is that strong contracts limit code reuse, while weak contracts limit reasoning. When defining a contract, one needs to consider the possible future code reuse of the method. This conflicts with the open world assumption. Another problem is that when reusing a class which only has a weak specification, one must look at the actual code to find out what the class does.

A recent line of work investigates how to control reasoning with an open

world assumption, with less severe restrictions than behavioral subtyping. *Lazy behavioral subtyping* proposes a lazy approach to behavioral subtyping's restriction to property preservation to a "restrict-by-need" incremental program development, which enables more programs to be verified than with behavioral subtyping. This is achieved by means of a book-keeping machinery to track the actually required properties in the inheritance tree. Lazy behavioral subtyping techniques were first developed for code reuse based on single class inheritance [11], and later also for multiple inheritance [12]. A similar book-keeping approach has been developed for refactoring [13]; in this setting verification is no longer fully incremental in the general case, but the book-keeping tracks which proofs are actually violated by a given change to the code and which proofs remain valid. A related line of research introduces *two-stage verification* (akin to incremental type-checking) by accepting that local reasoning is limited by later reuse and leaving some proof obligations to be instantiated in a precise manner in a second stage of reasoning. A proof system for traits has been developed following this approach [8]. *Uninterpreted predicates* make the unknown parts of this verification process explicit by representing unknown predicates in the specifications as variables, which are resolved in the second stage. This idea was introduced by the authors to establish the first modular proof system for delta-oriented programming [9]. *Abstract contracts* [5, 16] introduce similar variables in JML-like contracts to reason about class inheritance in KeY. This work also demonstrates that this kind of approach significantly reduces the verification effort by enabling proof reuse.

This paper develops a modular proof system based on uninterpreted predicates and a two-stage verification process for Lightweight Java [32], a subset of Java. We show that this proof system is sound and relative complete [7] with respect to a standard proof system for the same language, without uninterpreted predicates. We further show that the proposed framework can be applied to different code reuse mechanisms, such as class inheritance, trait-based programming, and delta-oriented programming. Thus, this paper extends our previous work [9] on a proof system for delta-oriented programs by providing a formalization of the proof system, soundness and relative completeness proofs for this proof system, and by relating it to class inheritance and to traits. Whereas previous related work introduces notions of soundness in terms of *internal consistency* (e.g., [5, 8, 11–13]), we are not aware of similar proofs of soundness and of relative completeness for this kind of proof system.

The paper is organized as follows. Section 2 presents the Lightweight Java language and its standard proof system. Section 3 introduces the proof system using uninterpreted predicates for code reuse. Section 4 establishes the soundness and completeness proofs for the proof system. Section 5 presents two different applications of the approach: delta-oriented programming and trait-based programming. Section 6 discusses related work and Section 7 concludes the paper.

CD	::= **class** C $\{\overline{\mathsf{FD}}; \overline{\mathsf{MD}}\}$	classes
FD	::= N f = c	fields
s	::= w = rhs | **if** (e) s **else** s **fi**	statements
rhs	::= e | **new** C | m($\overline{\mathsf{e}}$) | v.m($\overline{\mathsf{e}}$)	assignment rhs
N	::= C | **int** | **boolean**	nominal types
MD	::= N m($\overline{\mathsf{N}'\ \mathsf{x}}$){$\overline{\mathsf{N}''\ \mathsf{y}}$; $\overline{\mathsf{s}}$;**return** e;}	methods
w	::= f | y	assignable variables
v	::= w | x	program variables
e	::= v | this | op($\overline{\mathsf{e}}$)	expressions

Figure 1: The syntax of LWJ. Variable C is a class name, f is a field name, c is a constant (including null), x is a method parameter name, y is a name for a method local variable, and e are side-effect free expressions including primitive values.

2. The Language and Proof System of Lightweight Java

Lightweight Java (LWJ) [32] is a core calculus for an imperative subset of Java. Compared to Featherweight Java (FJ) [18], a major difference is that LWJ also models state whereas FJ is purely functional. We are focusing on modular reasoning with code reuse related to methods, and thus method binding is affected. To keep the language small, we therefore avoid other mechanisms such as loop constructs and avoid remote access to fields, which complicate modular reasoning. We do allow remote access to objects by means of method calls. The syntax of LWJ is shown in Figure 1 and briefly explained as follows. LWJ contains classes CD, fields FD, and statements s for assignment, conditionals, object creation, and method calls. LWJ has a nominal type system with types N which include class names, integers, and Booleans. Method definitions MD contains method parameters, local variable declarations, method body, a return statement, and method return type. The grammar distinguishes variables in assignable positions w from those in non-assignable positions v. Although single inheritance is supported by LWJ, it is not considered at this point in this paper. This is because flattened products written in LWJ will not have a class hierarchy. As usual overline notation denotes collections such as lists or sets, depending on the context; for instance, $\overline{\mathsf{s}}$ denotes a list of statements.

2.1. The Assertion Language

Let us now consider programs in LWJ for which methods have associated method specifications; i.e., class definitions have annotated method declarations AMD instead of the method declarations MD of Figure 1. The BNF syntax for annotated method declarations (AMD) is shown in Figure 2. Here, $\overline{\mathsf{sp}}$ is a list of method contracts and $\overline{\mathsf{req}}$ is a set of assumptions about other methods. The intuition is that method specifications $\overline{\mathsf{ap}}$ are guaranteed by the current method, assuming that the requirements $\overline{\mathsf{req}}$ are satisfied by the methods invoked by the current method. An method specification ap is a pair of *Boolean* assertions (predicates). Assertions a are side-effect free formulae, including expressions e, logical variables z, a reserved variable result for the method's return value, and

AMD	$::= \mathsf{MD}\ \overline{\mathsf{sp}}$	annotated method declarations
sp	$::= \mathbf{guar}\ \{\overline{\mathsf{ap}}\}\ \mathbf{req}\ \{\overline{\mathsf{req}}\}$	method contracts
ap	$::= (\mathsf{a}, \mathsf{a})$	method specification
req	$::= \mathsf{m} : \{\overline{\mathsf{ap}}\}$	requirements
a	$::= \mathsf{e} \mid \mathsf{z} \mid \mathsf{result} \mid \mathsf{op}(\overline{\mathsf{a}})$	assertions

Figure 2: The assertion language for LWJ.

compound expressions $\mathsf{op}(\overline{\mathsf{a}})$ which apply an operator op to a list of assertions. Note that method parameters are not assignable and the variable result can only be used in the postcondition and not in the precondition.

Basic notation. Let p and q be variables ranging over Boolean assertions, and let $p^{\overline{\mathsf{z}}}_{\overline{\mathsf{z}'}}$ denote that all occurrences of variables $\overline{\mathsf{z}}$ in p are replaced by the corresponding variable in $\overline{\mathsf{z}'}$. Primed variables are conventionally of the same syntactic category as the corresponding unprimed variables; for example, in $p^{\mathsf{y}}_{\mathsf{y}'}$ the program variable y is replaced by another program variable y'. This kind of renaming will typically be related to change of scope in the proof system in the sequel.

A Hoare triple $\{p\}\ \overline{\mathsf{s}}\ \{q\}$ expresses partial correctness [4]: if a statement list $\overline{\mathsf{s}}$ is executed in a state where assertion p holds and the execution terminates, then assertion q holds in the state upon termination. A method specification (p, q) for a method with body $\overline{\mathsf{s}}$ holds if the Hoare triple $\{p\}\ \overline{\mathsf{s}}\ \{q\}$ is valid. We consider proof systems concerning judgements of the form

$$\Gamma \vdash \{p\}\ \overline{\mathsf{s}}\ \{q\}$$

where Γ is a set of assumptions about such method specifications; these are written as $\mathsf{m}(\overline{\mathsf{N}\ \mathsf{x}}) : (p, q)$. These assumptions are used to make derivations about method calls in the statement list $\overline{\mathsf{s}}$.

Method Contracts. A method contract $\mathbf{guar}\ \{S\}\ \mathbf{req}\ \{\Gamma\}$, where S is a set of method specifications (p, q) and Γ is a list of requirements concerning the invoked methods, expresses that the specifications in S will hold for the method under the assumption that requirements in Γ hold for the invoked methods. The requirements in Γ are needed to decouple the verification of a method from the verification of the methods it calls. Thus, a method declaration for a method m has the form

$$\mathsf{N}\ \mathsf{m}(\overline{\mathsf{N}'\ \mathsf{x}})\{\overline{\mathsf{N}''\ \mathsf{y}}; \overline{\mathsf{s}}; \mathbf{return}\ \mathsf{e}\} \quad \mathbf{guar}\ \{S\}\ \mathbf{req}\ \{\Gamma\}.$$

The method contract expresses that

$$\forall (p, q) \in S \cdot \Gamma \vdash \{p^{\overline{\mathsf{z}}}_{\overline{\mathsf{z}'}}\}\ \overline{\mathsf{s}}; \mathbf{return}\ \mathsf{e}\ \{q^{\overline{\mathsf{z}}}_{\overline{\mathsf{z}'}}\},$$

where the logical variables $\overline{\mathsf{z}}$ are renamed to fresh names $\overline{\mathsf{z}'}$ to avoid name capture between different scopes. Thus, for each method specification $(p, q) \in S$, we need to prove the corresponding Hoare triple.

The set S of method specifications may contain multiple specifications for the same method. One motivation for allowing multiple specifications, is to keep each specification simple and easy to understand, as in the example of Section 2.1.1 below. Multiple specifications can also be used to allow concrete specification alongside abstract specifications with uninterpreted symbols, when uninterpreted symbols are added in Section 3, as illustrated by the example in Section 3.1.1. The requirements in Γ may similarly provide multiple specifications for the same method, for the same reasons. This also allows requirement specifications to be collected from different modules.

We may express that a program variable w is not changed by a method using the specification $(w = z, w = z)$ where z is a logical variable. For convenience, this kind of specification will be abbreviated to **readonly** $\overline{w}$ in the sequel (generalized to one or more variables). The use of this assertion restricts the set of assignable variables and is critical in the rely-guarantee setting as explained in [9]: It implicitly defines the set of program variables that may be changed by a call (i.e., $\overline{\mathsf{w}}$ in the rule).

2.1.1. Bank Example

To motivate the use of uninterpreted predicates, we consider an example of different versions of a bank account in LWJ, using class inheritance. We first provide a LWJ version of the example with annotated method declarations in Figure 3. Consider a simple banking system where the interface IAccount defines the visible methods, namely deposit and withdraw. The functionalities of the deposit and withdraw methods will be realized by the update method, with different redefinitions in different subclasses. Class CBase provides the basic functionalities of methods update, deposit, and withdraw. In subclass CFee, the update method is redefined such that a fee is associated with each withdraw. The subclass CLimit extends class CFee and constrains the minimum amount that an account's balance must have. We use the notation **super**.m in a subclass to refer to method m as defined in the superclass.

In class CBase the call to update is late bound, so we do not know at reasoning time which definition of update the code is referring to. Thus we cannot characterize the behavior of such a late bound call by a (concrete) specification; however, we may state a minimal requirement such as

$$(\mathsf{bal}{=}{=}\mathsf{bal'},\ !\mathsf{result} \Rightarrow \mathsf{bal}{=}{=}\mathsf{bal'})$$

to express that the balance is not changed when the method is unsuccessful (i.e., when the method returns false). Here bal' is a logical variable used to capture the initial value of bal. Reasoning about late bound calls to update can then be based on this minimal requirement of update. This explains the requirements used in methods withdraw and deposit. For this style of reasoning, one should be clear about which specifications define minimal requirements (by some suitable syntax), and for each definition of update it must be shown that the minimal requirements follow from the given specification of that specific definition.

Note that the calls of form **super**.update are not late bound, so in the subclasses CFee and CLimit the required specifications of the update methods are

```
interface IAccount {
  boolean deposit(int x);
  boolean withdraw(int x);
}

class CBase implements IAccount {
  int bal = 0;

  boolean update(int x){
    bal = bal + x; return true;
  } guar{(bal==bal', result ∧ bal==bal'+x)}

  boolean deposit(int x){
    boolean b=false; if(x⩾0){b=update(x);} return b;
  } guar{(bal==bal', !result ⇒ bal==bal')}
    req{update:{(bal==bal', !result ⇒ bal==bal')}}

  boolean withdraw(int x){
    boolean b=false; if(x⩾0){b=update(−x);} return b;
  } guar{(bal==bal', !result ⇒ bal==bal')}
    req{update: {(bal==bal', !result ⇒ bal==bal')}}
}

class CFee extends CBase {
  int fee = 1;

  boolean update(int x){
    boolean b;
    if(x<0){b=super.update(x−fee);}
    else{b=super.update(x);}
    return b;
  } guar{ (x<0 ∧ bal==bal', result ∧ bal==bal'+(x−fee)),
          (x⩾0 ∧ bal==bal', result ∧ bal==bal'+x),
          readonly fee}
    req{super.update: {(bal==bal', result ∧ bal==bal'+x)}}
}

class CLimit extends CFee {
  int limit = 0;

  boolean update(int x){ boolean b = false;
    if(bal+x>limit){b=super.update(x);} return b;
  } guar{ (bal+x⩽limit ∧ bal==bal', !result ∧ bal==bal'),
          (bal+x>limit ∧ x<0 ∧ bal==bal', result ∧ bal==bal'+(x−fee)),
          (bal+x>limit ∧ x⩾0 ∧ bal==bal', result ∧ bal==bal'+x),
          readonly limit}
    req{super.update: {(x<0 ∧ bal==bal', result ∧ bal==bal'+(x−fee)),
                       (x⩾0 ∧ bal==bal', result ∧ bal==bal'+x)}}
}
```

Figure 3: A version of the Bank example without use of uninterpreted predicates.

the same as the guaranteed specification of the update method in the corresponding superclass. In class CBase, the deposit and withdraw methods have more than one method specification in the guarantee clause. This is done for convenience, to make each specification simple by letting the two assertion pairs have complementary preconditions.

2.2. A proof system for LWJ

A basic proof system for LWJ is given in Figure 4. We conventionally express judgements in this proof system as $\Gamma \vdash_{\mathrm{LWJ}} \{p\}\ s\ \{q\}$ (with a subscript LWJ on the turnstile). Rules LWJ-assign, LWJ-skip, LWJ-new, and LWJ-return are axioms for side-effect free assignments, skip, object creation, and return statements, respectively. In Rule LWJ-new, $\mathit{fresh}(o)$ expresses that o is fresh in the sense that no program variable equals o.

The inference rule LWJ-implication allows preconditions to be strengthened and postconditions to be weakened. The rules LWJ-ifElse and LWJ-composition are for conditional statements and sequential composition, respectively. Rules LWJ-internal and LWJ-external are for internal and external method invocations. In these two rules, the formal parameters $\overline{\mathsf{x}}$ in the assertions p and q are substituted with actual parameters $\overline{\mathsf{e}}$, the variable result is substituted by the method's return value w, and the variable this is substituted by the object identity v.

Observe that the substitution of fresh names in the same syntactic category is used to avoid name capture between scopes; e.g., $\overline{\mathsf{w}}'$ and $\overline{\mathsf{w}}''$ replace the names of program variables $\overline{\mathsf{w}}$ in the pre- and postconditions of LWJ-external. For internal calls we can make assumptions about the value of fields, for external calls we can not make assumptions about fields. This renaming is justified by our target of a modular reasoning system. LWJ has been chosen such that its proof system is sound and relative complete:

Theorem 1 (Soundness of LWJ proof system)**.** *The proof system for LWJ is sound, i.e.,* $\Gamma \vdash_{LWJ} \{p\}\ \overline{\mathsf{s}}\ \{q\} \Rightarrow \Gamma \models_{LWJ} \{p\}\ \overline{\mathsf{s}}\ \{q\}$.

Proof. The proof follows from the related proof in [4], adapted to LWJ. □

Theorem 2 (Relative completeness of LWJ proof system)**.** *The proof system for the LWJ is relative complete, i.e.,* $\Gamma \models_{LWJ} \{p\}\ \overline{\mathsf{s}}\ \{q\} \Rightarrow \Gamma \vdash_{LWJ} \{p\}\ \overline{\mathsf{s}}\ \{q\}$.

Proof. The proof follows from the related proof in [10], adapted to LWJ. □

3. Modular Reasoning With Uninterpreted Predicates

A general rely-guarantee technique which can deal with compositional reasoning for delta-oriented programs is presented in [9]. This approach introduces so-called *uninterpreted predicates*; i.e., symbolic names are used as placeholders for the pre- and postconditions of methods for which method binding cannot be decided at reasoning time. The pre- and postconditions of a defined method with calls to context-dependent methods, may contain occurrences of such uninterpreted predicates. We now develop such an approach for LWJ in terms of a new proof system denoted UJ.

LWJ-assign
$$\Gamma \vdash_{\mathrm{LWJ}} \{q^{\mathsf{w}}_{\mathsf{e}}\}\ \mathsf{w} = \mathsf{e}\ \{q\}$$

LWJ-skip
$$\Gamma \vdash_{\mathrm{LWJ}} \{q\}\ \mathbf{skip}\ \{q\}$$

LWJ-new
$$\Gamma \vdash_{\mathrm{LWJ}} \{\forall o \cdot \mathit{fresh}(o) \Rightarrow q^{\mathsf{w}}_{o}\}\ \mathsf{w} = \mathbf{new}\ \mathsf{C}\ \{q\}$$

LWJ-return
$$\Gamma \vdash_{\mathrm{LWJ}} \{q^{\mathsf{result}}_{\mathsf{e}}\}\ \mathbf{return}\ \mathsf{e}\ \{q\}$$

LWJ-implication
$$\frac{p' \Rightarrow p \quad \Gamma \vdash_{\mathrm{LWJ}} \{p\}\bar{\mathsf{s}}\{q\} \quad q \Rightarrow q'}{\Gamma \vdash_{\mathrm{LWJ}} \{p'\}\bar{\mathsf{s}}\{q'\}}$$

LWJ-ifElse
$$\frac{\Gamma \vdash_{\mathrm{LWJ}} \{p \wedge e\}\ \overline{\mathsf{s}_1}\ \{q\} \qquad \Gamma \vdash_{\mathrm{LWJ}} \{p \wedge \neg e\}\ \overline{\mathsf{s}_2}\ \{q\}}{\Gamma \vdash_{\mathrm{LWJ}} \{p\}\ \mathbf{if}\ (\mathsf{e})\ \overline{\mathsf{s}_1}\ \mathbf{else}\ \overline{\mathsf{s}_2}\ \mathbf{fi}\ \{q\}}$$

LWJ-composition
$$\frac{\Gamma \vdash_{\mathrm{LWJ}} \{p\}\ \overline{\mathsf{s}_1}\ \{g\} \qquad \Gamma \vdash_{\mathrm{LWJ}} \{g\}\ \overline{\mathsf{s}_2}\ \{q\}}{\Gamma \vdash_{\mathrm{LWJ}} \{p\}\ \overline{\mathsf{s}_1}; \overline{\mathsf{s}_2}\ \{q\}}$$

LWJ-internal
$$\frac{(\mathsf{m}(\overline{\mathsf{N}\ \mathsf{x}}) : (p, q)) \in \Gamma}{\Gamma \vdash_{\mathrm{LWJ}} \{p^{\bar{\mathsf{x}},\bar{\mathsf{y}}}_{\bar{\mathsf{e}},\bar{\mathsf{y}}'}\}\ \mathsf{w} = \mathsf{m}(\bar{\mathsf{e}})\ \{q^{\bar{\mathsf{x}},\bar{\mathsf{y}},\mathsf{result}}_{\bar{\mathsf{e}},\bar{\mathsf{y}}'',\mathsf{w}}\}}$$

LWJ-external
$$\frac{(\mathsf{m}(\overline{\mathsf{N}\ \mathsf{x}}) : (p, q)) \in \Gamma}{\Gamma \vdash_{\mathrm{LWJ}} \{p^{\overline{\mathsf{w}},\,\mathsf{this},\,\bar{\mathsf{x}}}_{\overline{\mathsf{w}}',\,\mathsf{v},\,\bar{\mathsf{e}}}\}\ \mathsf{w} = \mathsf{v}.\mathsf{m}(\bar{\mathsf{e}})\ \{q^{\overline{\mathsf{w}},\,\mathsf{this},\,\bar{\mathsf{x}},\,\mathsf{result}}_{\overline{\mathsf{w}}'',\mathsf{v},\,\bar{\mathsf{e}},\,\mathsf{w}}\}}$$

Figure 4: Proof rules for LWJ.

3.1. The Assertion Language

The syntax for annotated method definitions for UJ is shown in Figure 5. Assertions $\mathsf{a_u}$ differ from the assertions a of Section 2 in that $\mathsf{a_u}$ may contain *uninterpreted predicates* u and explicit substitutions σ over assertions $\mathsf{a_u}$. Uninterpreted predicates u are conventionally capitalized (e.g., P and Q). Uninterpreted predicates play the role of placeholders in symbolic assumptions and represent the pre- and postconditions of methods where the exact specifications are unknown at reasoning time. It is easy to see that the assertions a in the proof system LWJ (language of Figure 2) form a subset of the annotations $\mathsf{a_u}$. An assertion which does not contain any occurrences of uninterpreted predicate variables, is called *concrete*.

Explicit substitutions σ, which bind assignable program variables w to assertions $\mathsf{a_u}$, are defined in terms of how they reduce to regular substitutions for concrete assertions.

Definition 3.1 (Explicit substitution)**.** Let a be a concrete assertion, $\mathsf{a_{u1}}$ and $\mathsf{a_{u2}}$ be assertions with uninterpreted predicates, w a program variable, and e an expression. Define explicit substitution inductively over assertions, as follows:

$$\mathsf{a}[\mathsf{w} := \mathsf{e}] \ \Leftrightarrow\ \mathsf{a}^{\mathsf{w}}_{\mathsf{e}}$$
$$(\mathsf{a_{u1}}\ \mathrm{op}\ \mathsf{a_{u2}})[\mathsf{w} := \mathsf{e}] \ \Leftrightarrow\ (\mathsf{a_{u1}}[\mathsf{w} := \mathsf{e}])\ \mathrm{op}\ (\mathsf{a_{u2}}[\mathsf{w} := \mathsf{e}]).$$

AMD	$::=$	$\mathsf{MD}\ \overline{\mathsf{sp}}$	annotated method declarations
sp	$::=$	$\mathbf{guar}\ \{\overline{\mathsf{ap}}\}\ \mathbf{req}\ \{\overline{\mathsf{req}}\}$	method contracts
ap	$::=$	$(\mathsf{a_u}, \mathsf{a_u})$	method specifications
req	$::=$	$\mathsf{m} : \{\overline{\mathsf{ap}}\}$	requirements
$\mathsf{a_u}$	$::=$	$\mathsf{e} \mid \mathsf{z} \mid \mathsf{result} \mid \mathsf{op}(\overline{\mathsf{a_u}}) \mid \mathsf{u} \mid \mathsf{a_u}\sigma$	assertions
u	$::=$	$\mathsf{P} \mid \mathsf{Q} \mid \ldots$	uninterpreted predicate variables
σ	$::=$	$[\overline{\mathsf{w}} := \overline{\mathsf{a}}]$	explicit concrete substitutions

Figure 5: Annotated method declarations for the UJ proof system.

Explicit substitution reduces to textual substitution for *concrete* assertions. Explicit substitution cannot be reduced for uninterpreted predicates. For multiple explicit substitutions with disjoint variables w and w', we have

$$(\mathsf{a_u}[\mathsf{w} := \mathsf{e}])[\mathsf{w}' := \mathsf{e}'] \quad \Leftrightarrow \quad \mathsf{a_u}[\mathsf{w}, \mathsf{w}' := \mathsf{e}^{\mathsf{w}'}_{\mathsf{e}'}, \mathsf{e}'].$$

(Otherwise, the substitutions must be done left-to-right; e.g., $\mathsf{a_u}[\mathsf{x} := \mathsf{e}_1][\mathsf{x} := \mathsf{e}_2]$ is $\mathsf{a_u}[\mathsf{x} := \mathsf{e}_2]$.) We now define ground substitutions for assertions $\mathsf{a_u}$ as follows:

Definition 3.2 (Ground substitution)**.** A *ground substitution* $\tau : \mathsf{u} \to \mathsf{a}$ is defined inductively over assertions $\mathsf{a_u}$ as follows:

$$\begin{aligned}
&\mathsf{a}\tau = \mathsf{a} \\
&\mathsf{op}(\overline{\mathsf{a_u}})\tau = \mathsf{op}(\overline{\mathsf{a_u}\tau}) \\
&\mathsf{u}\tau = \begin{cases} \mathsf{a} & \text{if } \tau(\mathsf{u}) = \mathsf{a} \\ \mathsf{u} & \text{otherwise} \end{cases} \\
&\mathsf{a_u}\sigma\tau = \mathsf{a_u}\tau\sigma
\end{aligned}$$

We say that a ground substitution *concretizes* an assertion $\mathsf{a_u}$ if $\mathsf{a_u}\tau$ is concrete. By extension, we will call a proof context Γ concrete if its method specifications are pairs of concrete assertions and we say that a substitution concretizes a proof context if it makes all assertions in the proof context concrete. We let Γ_τ denote the proof context Γ concretized by the ground substitution τ. The validity of assertions with uninterpreted predicates can now be defined in terms of the possible extensions into concrete assertions.

Definition 3.3 (Validity of assertions)**.** Let $\mathsf{a_u}$ be an assertion and τ be a ground substitution which concretizes $\mathsf{a_u}$. An assertion $\mathsf{a_u}$ is *valid* if and only if $\mathsf{a_u}\tau$ is valid for any such τ:

$$\models \mathsf{a_u} \ \Leftrightarrow \forall\tau \cdot \models \mathsf{a_u}\tau.$$

It is easy to see from Definition 3.3 that uninterpreted predicates are implicitly quantified over the entire formula; e.g., in the formula $\mathsf{P} \wedge \mathsf{Q} \Rightarrow \mathsf{P} \vee \mathsf{Q}$, the two occurrences of P denote two occurrences of the same formula, and similarly for Q.

```
interface IAccount {
  boolean deposit(int x);
  boolean withdraw(int x);
}

class CBase implements IAccount {
  int bal = 0;
  boolean update(int x){
    bal = bal + x; return true;
  } guar{(bal==bal', result ∧ bal==bal'+x)}

  boolean deposit(int x){
    boolean b=false; if(x⩾0){b=update(x);} return b;
  } guar{(x<0 ∧ bal==bal', !result ∧ bal==bal'),
          (x⩾0 ∧ P, Q)}
    req{update:{(P,Q)}}

  boolean withdraw(int x){
    boolean b=false; if(x⩾0){b=update(−x);} return b;
  } guar{(x<0 ∧ bal==bal', !result ∧ bal==bal'),
          (x⩾0 ∧ P[x:=−x], Q[x:=−x])}
    req{update: {(P,Q)}}
}
```

Figure 6: Class inheritance with uninterpreted predicates.

3.1.1. The Bank Example Revisited

A main complication in reasoning about classes and inheritance is the problem of late binding. A method call made in a class (textually speaking) may bind to a redefinition of the method belonging to a subclass that is designed later. For such a call there need not be a fixed contract that is known at the time when the enclosing method is defined.

We may therefore use the idea of uninterpreted predicates to express a *preliminary specification* for the call. The body of the defined method can be analyzed under the assumption of such specifications for each method m called in the body.

For a call to m where a concrete method specification is known at the time when the enclosing method is defined, one may use the concrete specification. However, one must then verify that each redefinition of m actually conforms with this fixed specification (as for behavioral subtyping [22]). For methods with specifications using uninterpreted predicates there is no such verification obligation, but reasoning results will in general depend on the uninterpreted predicates.

Figure 6 extends the banking example of Example 2.1.1 with assertions which make use of uninterpreted predicate variables. Here we only show the part of the code which differs from the previous version. The classes CFee and CLimit are as before, so only class CBase is presented.

The full class hierarchy is not yet known while reasoning about class CBase

and the calls to update in class CBase are late bound. Hence, the method contract of the deposit method contains uninterpreted predicates capturing a symbolic method specification of the unknown method update (i.e., update:{(P,Q)}). The withdraw method follows the same reasoning approach. The use of uninterpreted symbols allows us to postpone the decision of specifying specific properties of update. The instantiation of the uninterpreted predicates is postponed until the binding is known, allowing the reasoning system to be adaptable to any compilable implementations.

Note that we also define a concrete method specification

$$(\mathsf{x}{<}0 \wedge \mathsf{bal}{=}{=}\mathsf{bal}', \mathsf{!result} \wedge \mathsf{bal}{=}{=}\mathsf{bal}')$$

which expresses that the balance is not changed when the input value x is less than zero. This is a concrete specification for both methods deposit and withdraw under the condition x<0. Concrete method specifications may be used in contracts, alongside specifications with uninterpreted predicate symbols. In particular, the concrete specification and reasoning done in Section 2.1.1 can be included. Thus the use of UJ may be seen as an extension of behavioral subtyping that provides more flexibility. UJ guarantees that the Java system, no matter what the method binding is, will always satisfy the method specifications. Verification aspects are explained in Section 3.2.1.

3.2. The Proof System UJ

The proof system UJ is given in Figure 7. We conventionally express judgements in this proof system as $\Gamma \vdash_{\mathrm{UJ}} \{p\}\ \mathsf{s}\ \{q\}$ (with a subscript UJ on the turnstile). The UJ proof system resembles the LWJ proof system of Section 2, except that it uses explicit substitutions and assertions p and q may now contain uninterpreted predicates. The rules related to method invocations formalize how to reason about uninterpreted predicates, which was outlined but not formalized in [9]. When analyzing a given method body, the method specification (p, q) will be verified under the assumption of the requirements Γ of the method contract. We allow Γ to contain uninterpreted predicates, which represent preliminary method specifications for methods which are not yet known, as well as concrete specifications for methods which are known. The assertions p and q may contain uninterpreted predicates originating from the require clause Γ.

The proof rules in Figure 7 use explicit substitution because textual substitution is not possible on uninterpreted predicates. Observe that the substitutions in the rules for internal and external calls represent changes in scope and erase any knowledge of locally scoped variables. For rule UJ-internal, y' and y'' denote fresh program variables. Similarly to the assertion language of Section 2, the variable result can only be used in the postcondition. For simplicity, we do not give rules in the reasoning system that specifically deal with recursive calls.

3.2.1. Verification of the Bank Example

We now consider the verification of the bank example from Section 3.1.1. With the LWJ system and behavioral subtyping, we were not able to reason

UJ-assign
$$\Gamma \vdash_{\mathrm{UJ}} \{q[\mathsf{w} := \mathsf{e}]\}\ \mathsf{w} = \mathsf{e}\ \{q\}$$

UJ-skip
$$\Gamma \vdash_{\mathrm{UJ}} \{q\}\ \mathbf{skip}\ \{q\}$$

UJ-new
$$\Gamma \vdash_{\mathrm{UJ}} \{\forall o \cdot \mathit{fresh}(o) \Rightarrow q[\mathsf{w} := o]\}\ \mathsf{w} = \mathbf{new}\ \mathsf{C}\ \{q\}$$

UJ-return
$$\Gamma \vdash_{\mathrm{UJ}} \{q[\mathsf{result} := \mathsf{e}]\}\ \mathbf{return}\ \mathsf{e}\ \{q\}$$

UJ-implication
$$\frac{p' \Rightarrow p \quad \Gamma \vdash_{\mathrm{UJ}} \{p\}\bar{\mathsf{s}}\{q\} \quad q \Rightarrow q'}{\Gamma \vdash_{\mathrm{UJ}} \{p'\}\bar{\mathsf{s}}\{q'\}}$$

UJ-ifElse
$$\frac{\Gamma \vdash_{\mathrm{UJ}} \{p \wedge \mathsf{e}\}\ \overline{\mathsf{s}_1}\ \{q\} \qquad \Gamma \vdash_{\mathrm{UJ}} \{p \wedge \neg\mathsf{e}\}\ \overline{\mathsf{s}_2}\ \{q\}}{\Gamma \vdash_{\mathrm{UJ}} \{p\}\ \mathbf{if}\ (\mathsf{e})\ \overline{\mathsf{s}_1}\ \mathbf{else}\ \overline{\mathsf{s}_2}\ \mathbf{fi}\ \{q\}}$$

UJ-composition
$$\frac{\Gamma \vdash_{\mathrm{UJ}} \{p\}\ \overline{\mathsf{s}_1}\ \{g\} \qquad \Gamma \vdash_{\mathrm{UJ}} \{g\}\ \overline{\mathsf{s}_2}\ \{q\}}{\Gamma \vdash_{\mathrm{UJ}} \{p\}\ \overline{\mathsf{s}_1}; \overline{\mathsf{s}_2}\ \{q\}}$$

UJ-internal
$$\frac{(\mathsf{m}(\overline{\mathsf{N}\ \mathsf{x}}) : (p, q)) \in \Gamma}{\begin{array}{l}\Gamma \vdash_{\mathrm{UJ}} \{p[\overline{\mathsf{x}}, \overline{\mathsf{y}} := \overline{\mathsf{e}}, \overline{\mathsf{y}'}]\} \\ \quad \mathsf{w} = \mathsf{m}(\overline{\mathsf{e}})\ \{q[\overline{\mathsf{x}}, \overline{\mathsf{y}}, \mathsf{result} := \overline{\mathsf{e}}, \overline{\mathsf{y}''}, \mathsf{w}]\}\end{array}}$$

UJ-external
$$\frac{(\mathsf{m}(\overline{\mathsf{N}\ \mathsf{x}}) : (p, q)) \in \Gamma}{\begin{array}{l}\Gamma \vdash_{\mathrm{UJ}} \{p[\overline{\mathsf{w}}, \mathsf{this}, \overline{\mathsf{x}} := \overline{\mathsf{w}'}, \mathsf{v}, \overline{\mathsf{e}}]\} \\ \quad \mathsf{w} = \mathsf{v}.\mathsf{m}(\overline{\mathsf{e}})\ \{q[\overline{\mathsf{w}}, \mathsf{this}, \overline{\mathsf{x}}, \mathsf{result} := \overline{\mathsf{w}''}, \mathsf{v}, \overline{\mathsf{e}}, \mathsf{w}]\}\end{array}}$$

Figure 7: Proof rules for the UJ Language.

about late bound calls, apart from what follows from minimal requirements. The UJ proof system makes modular reasoning of code reuse possible. Compared to the LWJ proof system, the UJ proof system can verify each single method without knowing how the method binding will proceed at runtime. The proof can be reused under the various method bindings.

Using the UJ proof system for the bank example of Section 3.1.1, we can verify methods deposit and withdraw in class CBase without knowing which update method will actually be invoked by deposit and withdraw at runtime. However, in LWJ reasoning, specifications of all the methods should be concrete. Namely, the specifications of the deposit and withdraw methods in CBase in Section 2.1.1 are valid for all the possible method bindings for update at reasoning time. This is clearly challenging to achieve and therefore the specifications need to be very weak. Hexsre we show the proof sketch of the deposit method in CBase, which has one concrete specification and one with uninterpreted predicates P and Q.

- Method specification 1:
  {x<0 ∧ bal==bal'}
  boolen b=false;
  {x<0 ∧ bal==bal' ∧ !b}

if(x⩾0){b=update(x);}
{x<0 ∧ bal==bal' ∧ !b}
return b;
{!result ∧ bal==bal'}

- Method specification 2:
{x⩾0 ∧ P}
boolen b=false;
{x⩾0 ∧ P ∧ !b}
if(x⩾0){ {P ∧ !b}
b=update(x);}
{Q[result:=b]}
return b;
{Q}

This proof can be done by the KeY theorem prover [1], which records the proof of Java methods annotated with uninterpreted variables and concretizes the proof later by replacing the uninterpreted predicate variables with concrete method specifications. For example, the concretizedxs specifications of method deposit in CFee are shown below:

(x<0 ∧ bal==bal', !result ∧ bal==bal')
(x⩾0 ∧ bal==bal', result ∧ bal==bal'+x)

which are derived by replacing P and Q in the specifications of deposit with the guaranteed specifications of update in CFee. The first concrete specification expresses that the balance is not changed if the input data x is less than zero. The second one expresses that x amount of money is deposited to the balance when x is larger and equal to zero. The case for class CLimit is similar. Note that when the exact class of object o is not known, for example o.deposit, we may consider all possible implementation of deposit, given that the whole program is known.

4. Soundness and Completeness of UJ

4.1. Soundness of UJ

The UJ proof system differs from the basic proof system for LWJ in its use of uninterpreted predicate variables. Let $\vdash_{\mathrm{LWJ}}$ denote the proof system for LWJ restricted so that there are no uninterpreted predicate variables, i.e., neither assertions nor assumptions in Γ contain uninterpreted predicate symbols. For $\vdash_{\mathrm{LWJ}}$ we may then use textual substitution instead of explicit substitution.

We want to prove soundness of the UJ proof system, in the sense that provable results are valid results:

$$\Gamma \vdash_{\mathrm{UJ}} \{p\}\ \bar{\mathsf{s}}\ \{q\} \Rightarrow \Gamma \models_{\mathrm{UJ}} \{p\}\ \bar{\mathsf{s}}\ \{q\}$$

where $\models_{\text{UJ}}$ denotes validity of the UJ system. Assumptions with uninterpreted predicate variables need to be instantiated by concrete specifications such that no uninterpreted predicate variables are left. This can be captured by replacing assumption Γ to Γ_τ with some instantiation τ.

Definition 4.1 (Validity of UJ)**.** Let p and q range over assertions in UJ, Γ over proof contexts, and τ over ground substitutions which concretize p, q, and Γ. The *validity* of a judgment $\Gamma \vdash_{\text{UJ}} \{p\}\ \bar{\mathsf{s}}\ \{q\}$, written $\Gamma \models_{\text{UJ}} \{p\}\ \bar{\mathsf{s}}\ \{q\}$, is defined as follows:

$$\Gamma \models_{\text{UJ}} \{p\}\ \bar{\mathsf{s}}\ \{q\} \ \Leftrightarrow\ \forall \tau \cdot \Gamma_\tau \models_{\text{LWJ}} \{p\tau\}\ \bar{\mathsf{s}}\ \{q\tau\}.$$

Theorem 3. *Let p and q range over assertions in UJ, Γ over proof contexts, and τ over ground substitutions which concretize p, q, and Γ. Then*

$$\Gamma \vdash_{UJ} \{p\}\ \bar{\mathsf{s}}\ \{q\} \Rightarrow \forall \tau \cdot \Gamma_\tau \vdash_{LWJ} \{p\tau\}\ \bar{\mathsf{s}}\ \{q\tau\}.$$

Proof. The proof is by induction over the derivation in the UJ proof system. Details can be found in AppendixA. □

Soundness of the reasoning system for LWJ is stated in Theorem 1, from which we know

$$\Gamma_\tau \vdash_{\text{LWJ}} \{p\tau\}\ \bar{\mathsf{s}}\ \{q\tau\} \Rightarrow \Gamma_\tau \models_{\text{LWJ}} \{p\tau\}\ \bar{\mathsf{s}}\ \{q\tau\}$$

for Γ_τ, $p\tau$ and $q\tau$ without uninterpreted symbols.

Thus in order to prove soundness of $\vdash_{\text{UJ}}$ it suffices to prove

$$\Gamma \vdash_{\text{UJ}} \{p\}\ \bar{\mathsf{s}}\ \{q\} \Rightarrow \Gamma_\tau \vdash_{\text{LWJ}} \{p\tau\}\ \bar{\mathsf{s}}\ \{q\tau\} \quad \text{for any } \tau$$

for Γ_τ, $p\tau$, and $q\tau$ as above.

Theorem 4. *The UJ proof system is sound, i.e., $\Gamma \vdash_{UJ} \{p\}\ \bar{\mathsf{s}}\ \{q\} \Rightarrow \Gamma \models_{UJ} \{p\}\ \bar{\mathsf{s}}\ \{q\}$.*

Proof. The proof follows from Definition 4.1 and Theorems 1 and 3:

$$\begin{aligned}
&\Gamma \vdash_{\text{UJ}} \{p\}\ \bar{\mathsf{s}}\ \{q\} \\
&\qquad \Downarrow \text{ (by Theorem 3)} \\
&\forall \tau \cdot \Gamma_\tau \vdash_{\text{LWJ}} \{p\tau\}\ \bar{\mathsf{s}}\ \{q\tau\} \\
&\qquad \Downarrow \text{ (by Theorem 1)} \\
&\forall \tau \cdot \Gamma_\tau \models_{\text{LWJ}} \{p\tau\}\ \bar{\mathsf{s}}\ \{q\tau\} \\
&\qquad \Downarrow \text{ (by Definition 4.1)} \\
&\Gamma \models_{\text{UJ}} \{p\}\ \bar{\mathsf{s}}\ \{q\}
\end{aligned}$$

□

4.2. Relative Completeness of UJ

A system is complete if and only if all valid formulae can be derived from the axioms and the inference rules of the system. Completeness results for program verification systems are typically proven up to the completeness for underlying logic; this is called *relative completeness* (the notion is attributed to Cook [7]). We first prove that derivability in LWJ for all concretizations of a judgement in UJ entails derivability in UJ, from which the completeness result follows.

Theorem 5. *Let p and q range over assertions in UJ, Γ over proof contexts, and τ over ground substitutions which concretize p, q, and Γ. Then*

$$\forall \tau \cdot \Gamma_\tau \vdash_{LWJ} \{p\tau\}\ \bar{\mathsf{s}}\ \{q\tau\} \Rightarrow \Gamma \vdash_{UJ} \{p\}\ \bar{\mathsf{s}}\ \{q\}.$$

Proof. The proof is by induction over the derivation in the LWJ proof system. Details can be found in AppendixB. □

Theorem 6. *The UJ proof system is complete, i.e.,* $\Gamma \models_{UJ} \{p\}\ \bar{\mathsf{s}}\ \{q\} \Rightarrow \Gamma \vdash_{UJ} \{p\}\ \bar{\mathsf{s}}\ \{q\}$.

Proof. The proof follows from Definition 4.1 and Theorems 2 and 5:

$$\begin{aligned}
&\Gamma \models_{\mathrm{UJ}} \{p\}\ \bar{\mathsf{s}}\ \{q\} \\
&\qquad \Downarrow \text{ (by Definition 4.1)} \\
&\forall \tau \cdot \Gamma_\tau \models_{\mathrm{LWJ}} \{p\tau\}\ \bar{\mathsf{s}}\ \{q\tau\} \\
&\qquad \Downarrow \text{ (by Theorem 2)} \\
&\forall \tau \cdot \Gamma_\tau \vdash_{\mathrm{LWJ}} \{p\tau\}\ \bar{\mathsf{s}}\ \{q\tau\} \\
&\qquad \Downarrow \text{ (by Theorem 5)} \\
&\Gamma \vdash_{\mathrm{UJ}} \{p\}\ \bar{\mathsf{s}}\ \{q\}
\end{aligned}$$

□

4.3. Adaptation from Multiple Specifications

A method may have requirements given by a set of assertion pairs describing different or complimentary aspects of the method. This is convenient when a method is called several times and the different calls require different assertion pairs, but also when the requirement set of a method contains both concrete assertion pairs and assertion pairs with uninterpreted symbols.

Reasoning about a method call may in general involve one or more assertion pairs. The discussion so far is based on reasoning where only one requirement is needed for each call. Below we explain briefly how this restriction can be lifted. The following adaptation rule gives that from a set of assertion pairs (p_i, q_i) for $1 \leqslant i \leqslant N$, we may for any assertion $\mathsf{a_u}$ derive an assertion pair with exactly $\mathsf{a_u}$ as postcondition:

$$\textsf{adaptation} \qquad \frac{\Gamma \vdash_{\mathrm{UJ}} \{p_i\}\ \bar{\mathsf{s}}\ \{q_i\} \quad \text{for } 1 \leqslant i \leqslant N}{\Gamma \vdash_{\mathrm{UJ}} \{\forall \bar{\mathsf{u}} \cdot \bigwedge_{1 \leqslant i \leqslant N} (\forall \overline{\mathsf{z}_i} \cdot p_i \Rightarrow q_i[\bar{\mathsf{w}} := \bar{\mathsf{u}}]) \Rightarrow \mathsf{a_u}[\bar{\mathsf{w}} := \bar{\mathsf{u}}]\}\ \bar{\mathsf{s}}\ \{\mathsf{a_u}\}}$$

where $\overline{\mathsf{u}}$ are fresh, $\overline{\mathsf{z}_i}$ the logical variables in (p_i, q_i), and $\overline{\mathsf{w}}$ variables which may be updated by the method. We assume that these variables can be statically determined, which is ensured when adaptation is applied at the method level. The rule allows us to combine the information embedded in several assertion pairs for the same method. A further discussion on reasoning with multiple assertion pairs is given in [**?**], including soundness and completeness issues.

5. Application to Code Reuse Mechanisms

To demonstrate the applicability of our framework, we consider how it can be applied to different mechanisms for code reuse. We have used class inheritance to explain the concepts in Sections 2 and 3 and applied the framework to a bank account example that provides variations of functionalities to withdraw from and deposit money to a bank account, as shown in Sections 2.1.1 and 3.1.1. In this section, we consider two other code reuse mechanisms: delta-oriented programming and traits. In contrast to class inheritance, these two mechanisms can be characterized by the fact that the static binding of a method call is not even known at verification time inside a given module. We will continue using the bank account as the running example to illustrate our reasoning approach.

In the basic implementation of method withdraw, there are no limitations to the amount of money one can withdraw from the account, meaning that the balance may become negative. In addition, no extra fees are attached to this operation. For the basic implementation of method deposit, any positive amount can be deposited to the bank account. Both the deposit and withdraw methods use an update method to modify the account's balance. In the subsequent sections, we discuss how the implementation of the method update, under different code reuse mechanisms, will change the behaviour of money transaction. For each mechanism, we first show the implementation of the example, then the specification of all the methods, and finally the resulting, flattened software program in LWJ, derived by compilation.

5.1. Delta-oriented Programming

We consider a core language for Delta-Oriented Programming (DOP) [28] using a subset of the DeltaJ language of [33], including Delta-related mechanisms and augmented with annotations.

A DOP software product line (SPL) is realized by a set of deltas encapsulating modifications to object-oriented programs. Deltas can add, remove, or modify classes. A particular program variant is obtained by applying a selected set of deltas to the empty program in a given order. Note that an internal call $\mathsf{w} = \textbf{original}(\overline{\mathsf{e}})$ denotes a call to the previous version of the current method, defined outside the enclosing Delta definition.

In DOP our bank example is expressed as a SPL where Base is a mandatory feature, while Fee and Limit are optional features. We assume that we are always reasoning over valid feature configurations.

```
feature Base, Fee, Limit
configurations Base
deltas
  [DBase]
  [DFee when Fee]
  [DLimit when Limit]
```

Here, the **when** clause specifies for which feature configuration the delta has to be applied; i.e., [DFee **when** Fee] implies that the delta DFee is applied when the feature Fee is selected.

The code base of the Bank Account SPL is given as follows:

```
delta DBase{
  adds class Account{
    int bal = 0; // the balance
    boolean update(int x){
      bal = bal + x; return true;}

    boolean deposit(int x){ // for increasing the balance
      boolean b = false ;
      if(x ⩾ 0){b = update(x);} return b;}

    boolean withdraw(int x){ // for decreasing the balance
      boolean b = false;
      if(x ⩾ 0){b = update(−x);} return b;}
  }
}

delta DFee{
  modifies Account{
    adds int fee = 1;
    modifies boolean update(int x){
      boolean b;
      if(x < 0){b = original(x−fee);}
      else{b = original(x);}
      return b;}
  }
}

delta DLimit{
  modifies Account{
    adds int limit = 0;
    modifies boolean update(int x){
      boolean b = false;
      if(bal+x > limit){b=original(x);}
      return b;}
  }
}
```

The basic functionality of delta DBase consists of the methods update, deposit, and withdraw. Delta DFee modifies the Account class by adding a new field fee and by modifying method update such that it invokes the previous version of method update by means of **original**. Similarly, DLimit modifies the Account class

by adding a new field limit and by modifying method update. A particular bank account product is derived by applying to DBase the optional deltas DLimit, DFee, or both in a given order.

We demonstrate how uninterpreted predicates can make the verification scalable and reusable by verifying each delta in isolation. The uninterpreted predicates are used as placeholders for the method contract of the invoked methods, which are unknown at reasoning time. Consider the following specifications for the deltas:

```
// Specifications of delta DBase:
update: guar{(bal==bal', result ∧ bal==bal'+x)}
deposit: guar{(x<0 ∧ bal==bal', !result ∧ bal==bal'),
              (x≥0 ∧ P, Q)}
         req{update:{(P,Q)}}
withdraw: guar{(x<0 ∧ bal==bal', !result ∧ bal==bal')
              (x≥0 ∧ P[x:=−x],Q[x:=−x])}
         req{update: {(P,Q)}}

// Specifications of delta DFee:
update: guar{(x<0 ∧ P[x:=x−fee], Q[x:=x−fee]),
              (x≥0 ∧ P,Q),
              readonly fee}
         req{original: {(P,Q)}}

// Specifications of delta DLimit:
update: guar{(bal==bal' ∧ bal+x<=limit, !result ∧ bal==bal')
              (bal+x>limit ∧ P,Q),
              readonly limit}
         req{original: {(P,Q)}}
```

Here, the contract of the **original** method in delta DFee is **original**: {(P,Q)}. Compared to class inheritance, the uninterpreted predicates of the **original** method cannot be instantiated at reasoning time. It is not known which assertions to substitute for P and Q; this depends on the ordering of deltas, which is unknown at reasoning time.

Consider two different products A and B. Product A is formed by applying the features in the order Base, Limit, and then Fee:

```
class Account {
  int bal = 0; // the balance
  int limit = 0;
  int fee = 1;

  boolean update''(int x){bal = bal + x; return true;}

  boolean update'(int x){
    boolean b = false;
    if(bal+x > limit){b=update''(x);} return b;}

  boolean update(int x){
     boolean b;
     if(x < 0){b = update'(x−fee);}
```

```
    else{b = update'(x);}
    return b;}

  boolean deposit(int x){ boolean b = false;
    if(x ⩾ 0){b = update(x);} return b;}

  boolean withdraw(int x){ boolean b = false;
    if(x ⩾ 0){b = update(−x);} return b;}
}
```

Product B is formed by applying the features in the order Base, Fee, and then Limit:

```
class Account {
  int bal = 0; // the balance
  int limit = 0;
  int fee = 1;

  boolean update''(int x){bal = bal + x; return true;}

  boolean update'(int x){
    boolean b;
    if(x < 0){b = update''(x−fee);}
    else{b = update''(x);}
    return b;}

  boolean update(int x){
    boolean b = false;
    if(bal+x > limit){b=update'(x);} return b;}

  boolean deposit(int x){ boolean b = false;
    if(x ⩾ 0){b = update(x);} return b;}

  boolean withdraw(int x){ boolean b = false;
    if(x ⩾ 0){b = update(−x);} return b;}
}
```

According to different application orders of the deltas DLimit and DFee, the **original** call is replaced by calls to different implementations of method update. Product A guarantees that the balance in the bank account is always larger or equal to the limit, even when withdrawing money requires extra charge of fee. However, the balance of the bank account in Product B may be lower than limit due to the extra fee by withdrawing money. This can be observed by the concretized specifications listed below, in which we only show the cases when money is successfully withdrawn.

- Product A:
  (x<0 ∧ bal+(x-fee)>limit ∧ bal==bal', result ∧ bal==bal'+(x-fee))

- Product B:
  (x<0 ∧ bal+x>limit ∧ bal==bal', result ∧ bal==bal'+(x-fee))

Note that the uninterpreted predicates reasoning approach is modular and not affected by the application order of deltas, although the output results of different products may vary. For example, the update method in DFee guarantees that the extra fee will be charged when withdrawing money, and there is no requirement for depositing money. This functionality is independent of whether a minimum amount of balance is required or not.

5.2. Traits

Traits are so-called *pure units of behavior* [14, 30]: A trait contains a set of methods that can be used to extend the functionality of a class. Traits are completely independent from any class hierarchy and can be composed in an arbitrary order. In case of a naming collision, when multiple traits to be used by a class have methods with the same name, the programmer must explicitly disambiguate which of those methods will be used in the class (e.g., by renaming or deleting conflicting method definitions); thus manually solving the diamond problem of multiple inheritance.

The same bank account example can be implemented as follows using traits:

```
interface IAccount {
  boolean deposit(int x);
  boolean withdraw(int x);
}

trait TFunc is {
  boolean update(int x); // required method
  boolean deposit(int x){
    boolean b=false;
    if(x⩾0){b=update(x);} return b;}

  boolean withdraw(int x){
    boolean b=false;
    if(x⩾0){b=update(−x);} return b;}
}

trait TBase is {
  int bal = 0; // required field
  boolean update(int x){
    bal = bal + x; return true;}
}

trait TFee is {
  int bal = 0; int fee = 1; // required fields
  boolean update'(int x); // required method
  boolean update(int x){
    boolean b = false;
    if(x<0){b=update'(x−fee);}
    else{b=update'(x);} return b;}
}

trait TLimit is {
  int limit = 0; // required field
```

```
  boolean update'(int x); // required method
  boolean update(int x){
    boolean b = false;
    if(bal+x>limit){b=update'(x);}
    return b;}
}
```

Here, the trait TFunc provides the basic implementation of deposit and withdraw. The update method, invoked by methods deposit and withdraw in TFunc, is implemented in traits TBase, TFee, and TLimit. Depending on the selected combination of these traits, the derived class provides different functionalities of update.

Let us now consider the following specifications of traits:

```
// Specifications of trait TFunc:
deposit: guar{(x<0 ∧ bal==bal', !result ∧ bal==bal'),
              (x⩾0 ∧ P, Q)}
         req{update:{(P,Q)}}
withdraw: guar{(x<0 ∧ bal==bal', !result ∧ bal==bal')
              (x⩾0 ∧ P[x:=−x],Q[x:=−x])}
         req{update: {(P,Q)}}

// Specifications of trait TBase:
update: guar{(bal==bal', result ∧ bal==bal'+x)}

// Specifications of trait TFee:
update: guar{(x<0 ∧ P[x:=x−fee], Q[x:=x−fee]),
              (x⩾0 ∧ P,Q),
              readonly fee}
         req{update': {(P,Q)}}

// Specifications of trait TLimit:
update: guar{(bal==bal' ∧ bal+x<=limit, !result ∧ bal==bal')
              (bal+x>limit ∧ P,Q),
              readonly limit}
         req{update': {(P,Q)}}
```

Observe that the specifications are almost identical to the ones for deltas in Section 5.1 , except that the **original** method has been renamed to update' in the case of traits. The uninterpreted predicates of the update' method cannot be instantiated at reasoning time, but our reasoning system guarantees that for any compilable trait-based program, the program will satisfy the specification.

Below we show the definition of one possible trait-based class:

```
class CLimitFeeAcc implements IAccount
   by {int bal; int fee; int limit;} and TFunc +
        (TBase[update renameTo update'']) +
        (TFee[update renameTo update'][update' renameTo update'']) +
        TLimit
```

where class CLimitFeeAcc is formed by the combination of traits TFunc, TBase, TFee, and TLimit. In class CLimitFeeAcc, the update method is exactly the same

as the one in TLimit which invokes the update method in TFee that finally invokes the update method in TBase. For this purpose we keep the latest version of the update method (i.e., the one in TLimit) unchanged, and rename the other update methods according to the invocation order. Namely, in TFee the update and update' methods are renamed to update' and update", respectively, and in TBase the update method is renamed to update". The flattened class CLimitFeeAcc is shown below:

```
class CLimitFeeAcc implements IAccount {
  int bal = 0; int fee = 1; int limit = 0;
  boolean update''(int x){
    bal = bal + x; return true;}

  boolean update'(int x){
    boolean b=false;
    if(x<0){b=update''(x−fee);}
    else{b=update''(x);}
    return b;}

  boolean update(int x){
    boolean b = false;
    if(bal+x>limit){b=update'(x);}
    return b;}

  boolean deposit(int x){
    boolean b=false;
    if(x⩾0){b=update(x);} return b;}

  boolean withdraw(int x){
    boolean b=false;
    if(x⩾0){b=update(−x);} return b;}
}
```

This is exactly the same as the class Account of Product A in Section 5.1 derived by applying the deltas in the order DBase, DFee, and then DLimit.

6. Related Work

Modular reasoning of software components communicating with each other by method calls can be achieved by contract-based verification [23]. The work of [15] provides modular verification for delta-oriented programming based on Liskov's principle [22] in which the new method contracts must be substitutable for the old ones or, equivalently, only code changes that respect the existing contract are permitted. However, this is too restrictive in practice because already very simple code modifications tend to break existing contracts. In separation logic, abstract predicates [24] are defined as abbreviations for specific properties to enforce modular reasoning in the sense of information hiding. Inside a module the definition of the abstract properties can be expanded, while outside a module, only the abstract predicates can be used. The semantics of Java is not oriented towards modular reasoning. However, the principle of behavioral

subtyping, minimal requirements, abstract predicates, and ghost variables [**?**], can be applied to somewhat restricted subsets of Java, as well as techniques for refactoring [**?**]. For instance, the Java verifier of the KeY system [1] is based on inheritance of specifications from superclasses, thereby following behavioral subtyping.

The reasoning approach in [9] allows each delta module in delta-oriented programming to be verified in isolation, based on symbolic assumptions for calls to methods which may be defined in other delta modules. When product variants are generated from delta modules, these assumptions are instantiated by the actual guarantees of the method in the considered product variant. Hähnle et al. introduce abstract method calls for code reuse [16], using JML style specification [20]. The JML specification for each method is divided into two sections. The *abstract section* consists of placeholders for pre-and postconditions and assignable clauses. Each placeholder must be defined by concrete formulas and terms in the *definitions section*. This approach is implemented in the KeY theorem prover [1] by Pelevina [6] . The proof for calls to unknown methods using abstract contracts can be partially proved by KeY; i.e., proof branches cannot be completely closed due to the use of placeholders, but they can be cached for proof reuse. Once the invoked method is known, the abstract predicates can be instantiated with concrete predicates, and the symbolic execution in KeY can continue from the open goals of the cached partial proof. Therefore, the partial proofs can be reused under the context of variant method bindings. Proof replay reduces the verification effort. The implementation of the framework of proof repositories [5] in KeY improves the reusability and efficiency of proof reuse. It is well suited for object-oriented inheritance and late binding.

Compard to this line of work, our contribution is a generalization which considers not only class inheritance but also delta-oriented programming and traits. The calculus using uninterpreted predicates for modular reasoning can be used in all three kinds of structured code reuse. We provide a Hoare-style reasoning system based on uninterpreted predicates for the setting of LWJ, and prove soundness and completeness of this reasoning system. The paper [8] introduces a deductive proof system for traits-based object-oriented language. The proof approach is incremental. Compared with our work, the specification may be modified according to the need of code reuse and composition; however, reverification might be needed. A proof system for adaptable class hierarchies [13] is based on lazy behavioral subtyping. The proof system is incremental in the sense that reverification is avoided for methods that are not called by internal calls in the same class or subclass. But the reasoning for internal calls is based on minimal requirements that might be broken by modifications of these methods in a subclass. Compared with these previous works, our reasoning system is more flexible in the sense that we do not need reverification. An alternative approach to reasoning about object-oriented inheritance is based on separate hierarchies for code reuse, through class inheritance, and for inheritance of behavioral specification, through interfaces [**?** **?**]. This approach avoids the restrictions of behavioral subtyping and without enforcing minimal requirements, and avoids uninterpreted symbols. Invariants of superclasses may be violated in a subclass;

in such cases, reverification of inherited code is required.

Our system gives limited power for reasoning about external calls. In [13], reasoning about external calls are based on behavioral specifications in interfaces. The use of communication histories allows stronger method specifications than in our system. This approach can be applied and adapted to our setting as well.

7. Conclusion

Code reuse and modular software development greatly simplify the maintenance of complex software and shorten the development process until a new or enhanced release of a software program can be deployed. To support a development process based on flexible reuse and adaptation of software modules, it is crucial to be able to statically reason about the behavior of each module before composing reused modules into programs. However, standard software verification systems impose restrictions on software reuse and redefinition, such as behavioral subtyping. This paper has proposed a proof system which supports modular reasoning with an open-world assumption and which goes beyond behavioral subtyping. The proof system enables modular reasoning about languages with very flexible code reuse mechanisms in the context of object-oriented programming, as demonstrated in the paper by application to unrestricted class inheritance, delta-oriented programming, and trait-based programming. The approach taken is oriented towards class-wise verification. The LWJ system is compositional when contracts for external calls can be expressed in the assertion language. The UJ system improves on this by allowing modular reasoning based on symbolic contracts, to be instantiated at a later stage, possibly when the whole program is known. This means that with UJ one can verify each part of the software separately, based on the symbolic contracts. Interfaces and *communication histories* may be used to avoid talking about the overall system state, for instance in the style of [10]. Scalability of the contract instantiations depends on the chosen reuse mechanism. One possible approach to implement our reasoning system is to extend a weakest pre-condition calculus to generate assertions with uninterpreted predicate symbols and explicit substitutions, and using solver technologies such as SMT to find possible ground substitutions.

Since the environment of a piece of code is subject to change, the proposed proof system makes assumptions about the environment explicit in terms of uninterpreted predicates, which are instantiated in a second phase of reasoning (i.e., at compile time). Technically, we formulate the proof system with uninterpreted predicates for an object-oriented core language and relate the proposed proof system to a standard proof system for the same language. Whereas related work on incremental proof systems prove soundness in terms of internal consistency, this paper establishes both soundness and relative completeness of the proposed proof system by relating it to similar, established properties of a standard proof system for the same language.

Acknowledgement

We gratefully thank the anonymous reviewers for their constructive feedback.

References

[1] Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P. H., Ulbrich, M. (Eds.), 2016. Deductive Software Verification - The KeY Book - From Theory to Practice. Vol. 10001 of Lecture Notes in Computer Science. Springer.
URL http://dx.doi.org/10.1007/978-3-319-49812-6

[2] America, P., January 1989. A behavioural approach to subtyping in object-oriented programming languages. Tech. Rep. 443, Philips Research Laboratories.

[3] America, P., 1991. Designing an object-oriented programming language with behavioural subtyping. In: de Bakker, J. W., de Roever, W.-P., Rozenberg, G. (Eds.), Foundations of Object-Oriented Languages (REX Workshop). Vol. 489 of LNCS. Springer-Verlag, pp. 60–90.

[4] Apt, K. R., de Boer, F. S., Olderog, E., 2009. Verification of Sequential and Concurrent Programs. Texts in Computer Science. Springer.
URL http://dx.doi.org/10.1007/978-1-84882-745-5

[5] Bubel, R., Damiani, F., Hähnle, R., Johnsen, E. B., Owe, O., Schaefer, I., Yu, I. C., 2016. Proof repositories for compositional verification of evolving software systems - managing change when proving software correct. T. Foundations for Mastering Change 1, 130–156.
URL http://dx.doi.org/10.1007/978-3-319-46508-1_8

[6] Bubel, R., Hähnle, R., Pelevina, M., 2014. Fully abstract operation contracts. In: Margaria, T., Steffen, B. (Eds.), Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications - 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part II. Vol. 8803 of Lecture Notes in Computer Science. Springer, pp. 120–134.
URL http://dx.doi.org/10.1007/978-3-662-45231-8_9

[7] Cook, S. A., 1978. Soundness and completeness of an axiom system for program verification. SIAM J. Comput. 7 (1), 70–90.
URL https://doi.org/10.1137/0207005

[8] Damiani, F., Dovland, J., Johnsen, E. B., Schaefer, I., 2014. Verifying traits: an incremental proof system for fine-grained reuse. Formal Aspects of Computing 26 (4), 761–793.
URL http://dx.doi.org/10.1007/s00165-013-0278-3

[9] Damiani, F., Owe, O., Dovland, J., Schaefer, I., Johnsen, E. B., Yu, I. C., 2012. A transformational proof system for Delta-oriented programming. In: 16th International Software Product Line Conference, SPLC '12, Salvador, Brazil - September 2-7, 2012, Volume 2. ACM, pp. 53–60.
URL http://doi.acm.org/10.1145/2364412.2364422

[10] Din, C. C., Owe, O., 2014. A sound and complete reasoning system for asynchronous communication with shared futures. J. Log. Algebr. Meth. Program. 83 (5-6), 360–383.
URL http://dx.doi.org/10.1016/j.jlamp.2014.03.003

[11] Dovland, J., Johnsen, E. B., Owe, O., Steffen, M., 2010. Lazy behavioral subtyping. Journal of Logic and Algebraic Programming 79 (7), 578–607.

[12] Dovland, J., Johnsen, E. B., Owe, O., Steffen, M., 2011. Incremental reasoning with lazy behavioral subtyping for multiple inheritance. Science of Computer Programming 76 (10), 915–941.

[13] Dovland, J., Johnsen, E. B., Owe, O., Yu, I. C., 2015. A proof system for adaptable class hierarchies. Journal of Logical and Algebraic Methods in Programming 84 (1), 37 – 53, special Issue: The 23rd Nordic Workshop on Programming Theory (NWPT 2011) & Special Issue: Domains X, International workshop on Domain Theory and applications, Swansea, 5-7 September, 2011.
URL http://www.sciencedirect.com/science/article/pii/S2352220814000595

[14] Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R., Black, A. P., 2006. Traits: A mechanism for fine-grained reuse. ACM Transactions on Programming Languages and Systems 28 (2), 331–388.

[15] Hähnle, R., Schaefer, I., 2012. A Liskov principle for Delta-oriented programming. In: Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 5th International Symposium, 2012, Part I. Vol. 7609 of LNCS. Springer, pp. 32–46.
URL http://dx.doi.org/10.1007/978-3-642-34026-0_4

[16] Hähnle, R., Schaefer, I., Bubel, R., 2013. Reuse in software verification by abstract method calls. In: Bonacina, M. P. (Ed.), Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings. Vol. 7898 of Lecture Notes in Computer Science. Springer, pp. 300–314.
URL http://dx.doi.org/10.1007/978-3-642-38574-2_21

[17] Hähnle, R., Schafer, I., 2012. A Liskov principle for delta-oriented programming. In: International Conference on Formal Verification of Object-oriented Software (FoVeOOS 2011). Vol. 7421 of LNCS. Springer-Verlag, pp. 32–46.
URL https://doi.org/10.1007/978-3-642-34026-0_4

[18] Igarashi, A., Pierce, B. C., Wadler, P., 2001. Featherweight Java: a minimal core calculus for java and GJ. ACM Trans. Program. Lang. Syst. 23 (3), 396–450.
URL http://doi.acm.org/10.1145/503502.503505

[19] Leavens, G. T., Naumann, D. A., 2006. Behavioral subtyping, specification inheritance, and modular reasoning. Technical Report 06-20a, Department of Computer Science, Iowa State University, Ames, Iowa.

[20] Leavens, G. T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P., 2008. JML reference manual.

[21] Liskov, B., 1987. Data abstraction & hierarchy. In: Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'87). ACM, pp. 17–34, in *SIGPLAN Notices* 22(12).

[22] Liskov, B. H., Wing, J. M., Nov. 1994. A behavioral notion of subtyping. ACM Transactions on Programming Languages and Systems 16 (6), 1811–1841.

[23] Meyer, B., 1992. Applying "design by contract". IEEE Computer 25 (10), 40–51.
URL http://dx.doi.org/10.1109/2.161279

[24] Parkinson, M. J., Bierman, G. M., 2005. Separation logic and abstraction. In: Palsberg, J., Abadi, M. (Eds.), Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005. ACM, pp. 247–258.
URL http://doi.acm.org/10.1145/1040305.1040326

[25] Pierik, C., de Boer, F. S., 2005. A proof outline logic for object-oriented programming. *Theoretical Computer Science* 343 (3), 413–442.

[26] Poetzsch-Heffter, A., Müller, P., 1999. A programming logic for sequential Java. In: Swierstra, S. D. (Ed.), 8th European Symposium on Programming Languages and Systems (ESOP'99). Vol. 1576 of LNCS. Springer-Verlag, pp. 162–176.

[27] Schaefer, I., Bettini, L., Bono, V., Damiani, F., Tanzarella, N., 2010. Delta-oriented Programming of Software Product Lines. In: SPLC. Vol. 6287 of LNCS. Springer, pp. 77–91.

[28] Schaefer, I., Bettini, L., Bono, V., Damiani, F., Tanzarella, N., 2010. Delta-oriented programming of software product lines. In: Bosch, J., Lee, J. (Eds.), Software Product Lines: Going Beyond - 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13-17, 2010. Proceedings. Vol. 6287 of Lecture Notes in Computer Science. Springer, pp. 77–91.
URL http://dx.doi.org/10.1007/978-3-642-15579-6_6

[29] Schaefer, I., Bettini, L., Damiani, F., 2011. Compositional Type-Checking for Delta-Oriented Programming. In: AOSD 2011. ACM, pp. 43–56.

[30] Schärli, N., Ducasse, S., Nierstrasz, O., Black, A., 2003. Traits: Composable units of behavior. In: Proc. European Conference on Object-Oriented Programming (ECOOP). Vol. 2743 of LNCS. Springer-Verlag, pp. 248–274.

[31] Soundarajan, N., Fridella, S., 1998. Inheritance: From code reuse to reasoning reuse. In: Devanbu, P., Poulin, J. (Eds.), Proc. Fifth International Conference on Software Reuse (ICSR5). IEEE Computer Society Press, pp. 206–215.

[32] Strnisa, R., Sewell, P., Parkinson, M. J., 2007. The Java module system: core design and semantic definition. In: Gabriel, R. P., Bacon, D. F., Lopes, C. V., Jr., G. L. S. (Eds.), Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada. ACM, pp. 499–514.
URL http://doi.acm.org/10.1145/1297027.1297064

[33] http://deltaj.sourceforge.net/, 2011. DeltaJ website http://deltaj.sourceforge.net/.

AppendixA. Proof of Theorem 3

The proof is by induction over the construction of the $\vdash_{\mathrm{UJ}}$ proof; we show that for any such derivation we can construct corresponding derivations in LWJ. Thus, derivations by means of UJ-axioms constitute the base case and the induction hypothesis expresses the entailment of LWJ-derivability from UJ-derivability for the premisses of the UJ-rule. In the different cases of the proof, we let p and q range over assertions in UJ, Γ over proof contexts, and τ over ground substitutions which concretize p, q, and Γ.

Base Case. The base case covers the axioms of the inference system: UJ-assign, UJ-skip, UJ-new, and UJ-return. The proof proceeds by cases for each of the axioms.

Case UJ-assign. We consider the derivation of $\Gamma \vdash_{\mathrm{UJ}} \{q[\mathsf{w} := \mathsf{e}]\}\ \mathsf{w} = \mathsf{e}\ \{q\}$. By axiom LWJ-assign we know that

$$\forall \tau \cdot \Gamma_\tau \vdash_{\mathrm{LWJ}} \{q\tau^{\mathsf{w}}_{\mathsf{e}}\}\ \mathsf{w} = \mathsf{e}\ \{q\tau\},$$

Since the assertion $q\tau$ is concrete, we now have, by Definition 3.1, that

$$\forall \tau \cdot \Gamma_\tau \vdash_{\mathrm{LWJ}} \{q\tau[\mathsf{w} := \mathsf{e}]\}\ \mathsf{w} = \mathsf{e}\ \{q\tau\}.$$

and then, by Definition 3.2,

$$\forall \tau \cdot \Gamma_\tau \vdash_{\mathrm{LWJ}} \{q[\mathsf{w} := \mathsf{e}]\tau\}\ \mathsf{w} = \mathsf{e}\ \{q\tau\}.$$

Other base cases. The proofs for the axioms UJ-skip, UJ-new, and UJ-return follows the same pattern as the proof for rule UJ-assign.

Induction Step. For the induction step of the proof, we consider the inference rules UJ-implication, UJ-ifElse, UJ-composition, UJ-internal, and UJ-external. In these proof cases, we rely on an induction hypothesis (IH) expressing that $\Gamma \vdash_{\mathrm{UJ}} \{p\}\ \bar{\mathsf{s}}\ \{q\} \Rightarrow \forall \tau \cdot \Gamma_\tau \vdash_{\mathrm{LWJ}} \{p\tau\}\ \bar{\mathsf{s}}\ \{q\tau\}$ holds for any premisses $\{p\}\ \bar{\mathsf{s}}\ \{q\}$ of the inference rule.

Case UJ-implication. We have derived $\Gamma \vdash_{\mathrm{UJ}} \{p'\}\ \bar{\mathsf{s}}\ \{q'\}$ from the premisses $p' \Rightarrow p$, $\Gamma \vdash_{\mathrm{UJ}} \{p\}\bar{\mathsf{s}}\{q\}$, and $q \Rightarrow q'$. By the IH, we have

$$\forall \tau \cdot \Gamma_\tau \vdash_{\mathrm{LWJ}} \{p\tau\}\bar{\mathsf{s}}\{q\tau\}$$

and, by Definition 3.3, we know that $\forall \tau \cdot p'\tau \Rightarrow p\tau$ and $\forall \tau \cdot q\tau \Rightarrow q'\tau$. By applying LWJ-implication, we can conclude

$$\forall \tau \cdot \Gamma \vdash_{\mathrm{LWJ}} \{p'\tau\}\bar{\mathsf{s}}\{q'\tau\}.$$

Case UJ-ifElse. We have derived $\Gamma \vdash_{\mathrm{UJ}} \{p\}\ \mathbf{if}\ (\mathsf{e})\ \overline{\mathsf{s}_1}\ \mathbf{else}\ \overline{\mathsf{s}_2}\ \mathbf{fi}\ \{q\}$ from the premisses $\Gamma \vdash_{\mathrm{UJ}} \{p \wedge \mathsf{e}\}\ \overline{\mathsf{s}_1}\ \{q\}$ and $\Gamma \vdash_{\mathrm{UJ}} \{p \wedge \neg\mathsf{e}\}\ \overline{\mathsf{s}_2}\ \{q\}$, so, by the IH, we know that that $\forall \tau \cdot \Gamma_\tau \vdash_{\mathrm{LWJ}} \{(p \wedge \mathsf{e})\tau\}\ \overline{\mathsf{s}_1}\ \{q\tau\}$ and $\forall \tau \cdot \Gamma_\tau \vdash_{\mathrm{LWJ}} \{(p \wedge \neg\mathsf{e})\tau\}\ \overline{\mathsf{s}_2}\ \{q\tau\}$.

From Definition 3.2, we know that $\mathsf{e}\tau = \mathsf{e}$ and $\neg\mathsf{e}\tau = \neg\mathsf{e}$, so this is equivalent to $\forall\tau \cdot \Gamma_\tau \vdash_{\text{LWJ}} \{p\tau \wedge \mathsf{e}\}\ \overline{\mathsf{s}_1}\ \{q\tau\}$ and $\forall\tau \cdot \Gamma_\tau \vdash_{\text{LWJ}} \{p\tau \wedge \neg\mathsf{e}\}\ \overline{\mathsf{s}_2}\ \{q\tau\}$. We can now apply the rule LWJ-ifElse, to derive

$$\forall\tau \cdot \Gamma \vdash_{\text{LWJ}} \{p\tau\}\ \mathtt{if}\ (\mathsf{e})\ \overline{\mathsf{s}_1}\ \mathtt{else}\ \overline{\mathsf{s}_2}\ \mathtt{fi}\ \{q\tau\}.$$

Case UJ-composition. Here, p' is an assertion in UJ. We have derived $\Gamma \vdash_{\text{UJ}} \{p\}\ \overline{\mathsf{s}_1}; \overline{s_2}\ \{q\}$ from the premisses $\Gamma \vdash_{\text{UJ}} \{p\}\ \overline{\mathsf{s}_1}\ \{p'\}$ and $\Gamma \vdash_{\text{UJ}} \{p'\}\ \overline{\mathsf{s}_2}\ \{q\}$, so, by the IH, we know that $\forall\tau \cdot \Gamma_\tau \vdash_{\text{UJ}} \{p\tau\}\ \overline{\mathsf{s}_1}\ \{p'\tau\}$ and $\forall\tau \cdot \Gamma_\tau \vdash_{\text{UJ}} \{p'\tau\}\ \overline{\mathsf{s}_2}\ \{q\tau\}$. We can then apply LWJ-composition to conclude

$$\forall\tau \cdot \Gamma \vdash_{\text{LWJ}} \{p\tau\}\ \overline{\mathsf{s}_1}; \overline{\mathsf{s}_2}\ \{q\tau\}.$$

Case UJ-internal. Here, $\overline{y}'$ and $\overline{y}''$ are assumed to be fresh variable names. We have derived

$$\Gamma \vdash_{\text{LWJ}} \{p[\overline{y}, \overline{\mathsf{x}} := \overline{y}', \overline{\mathsf{e}}]\}\ \mathsf{w} = \mathsf{m}(\overline{\mathsf{e}})\ \{q[\overline{y}, \overline{\mathsf{x}}, \mathsf{result} := \overline{y}'', \overline{\mathsf{e}}, \mathsf{w}]\}$$

from the premise $(\mathsf{m}(\overline{\mathsf{N}\ \mathsf{x}}) : (p, q)) \in \Gamma$. Consequently, by applying substitituions, we know that $\forall\tau \cdot (\mathsf{m}(\overline{\mathsf{N}\ \mathsf{x}}) : (p\tau, q\tau)) \in \Gamma_\tau$. We can then apply LWJ-internal to derive

$$\forall\tau \cdot \Gamma_\tau \vdash_{\text{LWJ}} \{p\tau^{\overline{y},\overline{\mathsf{x}}}_{\overline{y}',\overline{\mathsf{e}}}\}\ \mathsf{w} = \mathsf{m}(\overline{\mathsf{e}})\ \{q\tau^{\overline{y},\overline{\mathsf{x}},\mathsf{result}}_{\overline{y}'',\overline{\mathsf{e}},\mathsf{w}}\}.$$

Since $p\tau$ and $q\tau$ are concrete, Definition 3.1 gives us

$$\forall\tau \cdot \Gamma_\tau \vdash_{\text{LWJ}} \{p\tau[\overline{y}, \overline{\mathsf{x}} := \overline{y}', \overline{\mathsf{e}}]\}\ \mathsf{w} = \mathsf{m}(\overline{\mathsf{e}})\ \{q\tau[\overline{y}, \overline{\mathsf{x}}, \mathsf{result} := \overline{y}'', \overline{\mathsf{e}}, \mathsf{w}]\}$$

and, from Definition 3.2, we get

$$\forall\tau \cdot \Gamma_\tau \vdash_{\text{LWJ}} \{p[\overline{y}, \overline{\mathsf{x}} := \overline{y}', \overline{\mathsf{e}}]\tau\}\ \mathsf{w} = \mathsf{m}(\overline{\mathsf{e}})\ \{q[\overline{y}, \overline{\mathsf{x}}, \mathsf{result} := \overline{y}'', \overline{\mathsf{e}}, \mathsf{w}]\tau\}.$$

Case UJ-external. Here, $\overline{w}'$ and $\overline{w}''$ are assumed to be fresh variable names. We have derived

$$\Gamma \vdash_{\text{UJ}} \{p[\overline{\mathsf{w}}, \mathsf{this}, \overline{\mathsf{x}} := \overline{\mathsf{w}}', \mathsf{v}, \overline{\mathsf{e}}]\}\ \mathsf{w} = \mathsf{v}.\mathsf{m}(\overline{\mathsf{e}})\ \{q[\overline{\mathsf{w}}, \mathsf{this}, \overline{\mathsf{x}}, \mathsf{result} := \overline{\mathsf{w}}'', \mathsf{v}, \overline{\mathsf{e}}, \mathsf{w}]\}$$

from the premise $(\mathsf{m}(\overline{\mathsf{N}\ \mathsf{x}}) : (p, q)) \in \Gamma$. By applying substitutions, we know that $\forall\tau \cdot (\mathsf{m}(\overline{\mathsf{N}\ \mathsf{x}}) : (p\tau, q\tau)) \in \Gamma_\tau$. We can then apply LWJ-external to derive

$$\forall\tau \cdot \Gamma_\tau \vdash_{\text{LWJ}} \{p\tau^{\overline{\mathsf{w}},\mathsf{this},\overline{\mathsf{x}}}_{\overline{\mathsf{w}}',\mathsf{v},\overline{\mathsf{e}}}\}\ \mathsf{w} = \mathsf{v}.\mathsf{m}(\overline{\mathsf{e}})\ \{q\tau^{\overline{\mathsf{w}},\mathsf{this},\overline{\mathsf{x}},\mathsf{result}}_{\overline{\mathsf{w}}'',\mathsf{v},\overline{\mathsf{e}},\mathsf{w}}\}.$$

Since $p\tau$ and $q\tau$ are concrete, Definition 3.1 gives us

$$\forall\tau \cdot \Gamma_\tau \vdash_{\text{LWJ}} \{p\tau[\overline{\mathsf{w}}, \mathsf{this}, \overline{\mathsf{x}} := \overline{\mathsf{w}}', \mathsf{v}, \overline{\mathsf{e}}]\}\ \mathsf{w} = \mathsf{v}.\mathsf{m}(\overline{\mathsf{e}})\ \{q\tau[\overline{\mathsf{w}}, \mathsf{this}, \overline{\mathsf{x}}, \mathsf{result} := \overline{\mathsf{w}}'', \mathsf{v}, \overline{\mathsf{e}}, \mathsf{w}]\}$$

and, from Definition 3.2, we get

$$\forall\tau \cdot \Gamma_\tau \vdash_{\text{LWJ}} \{p[\overline{\mathsf{w}}, \mathsf{this}, \overline{\mathsf{x}} := \overline{\mathsf{w}}', \mathsf{v}, \overline{\mathsf{e}}]\tau\}\ \mathsf{w} = \mathsf{v}.\mathsf{m}(\overline{\mathsf{e}})\ \{q[\overline{\mathsf{w}}, \mathsf{this}, \overline{\mathsf{x}}, \mathsf{result} := \overline{\mathsf{w}}'', \mathsf{v}, \overline{\mathsf{e}}, \mathsf{w}]\tau\}.$$

□

AppendixB. Proof of Theorem 5

The proof is by induction over the construction of the $\vdash_{\mathrm{LWJ}}$ proof; we show that for any such derivation we can construct a corresponding derivation in UJ. Thus, derivations by means of LWJ-axioms constitute the base case and the induction hypothesis expresses the entailment of UJ-derivability from LWJ-derivability for the premises of the LWJ-rule. In the different cases of the proof, we let p and q range over assertions in UJ, Γ over proof contexts, and τ over ground substitutions which concretize p, q, and Γ.

Base Case. The base case covers the axioms of the inference system: UJ-assign, UJ-skip, UJ-new, and UJ-return. The proof proceeds by cases for each of the axioms.

Case LWJ-assign. Assume that

$$\forall \tau \cdot \Gamma_\tau \vdash_{\mathrm{LWJ}} \{p\tau\}\ \mathsf{w} = \mathsf{e}\ \{q\tau\}.$$

From LWJ-assign, we know that

$$\forall \tau \cdot \Gamma_\tau \vdash_{\mathrm{LWJ}} \{q\tau^{\mathsf{w}}_{\mathsf{e}}\}\ \mathsf{w} = \mathsf{e}\ \{q\tau\}.$$

where $q\tau^{\mathsf{w}}_{\mathsf{e}}$ is the weakest possible precondition. Then, by LWJ-implication, relative completeness gives us

$$\forall \tau \cdot \vdash p\tau \Rightarrow q\tau^{\mathsf{w}}_{\mathsf{e}}$$

which, by Definitions 3.1 and 3.2, is equivalent to

$$\forall \tau \cdot \vdash p\tau \Rightarrow q[\mathsf{w} := \mathsf{e}]\tau$$

From soundness of first-order logic and Definition 3.3, we then get

$$\models p \Rightarrow q[\mathsf{w} := \mathsf{e}]$$

and, by relative completeness, $\vdash p \Rightarrow q[\mathsf{w} := \mathsf{e}]$. By UJ-assign we know that

$$\Gamma \vdash_{\mathrm{UJ}} \{q[\mathsf{w} := \mathsf{e}]\}\ \mathsf{w} = \mathsf{e}\ \{q\}$$

and UJ-implication of the UJ reasoning system, we derive

$$\Gamma \vdash_{\mathrm{UJ}} \{p\}\ \mathsf{w} = \mathsf{e}\ \{q\}.$$

Other base cases. The proofs for the axioms UJ-skip, UJ-new, and UJ-return follow the same pattern as the one for axiom UJ-assign.

Induction Step. When we consider the inference rules of LWJ, our induction hypothesis (IH) expresses that $\forall\tau \cdot \Gamma_\tau \vdash_{\text{LWJ}} \{p\tau\}\ \bar{\mathsf{s}}\ \{q\tau\} \Rightarrow \Gamma \vdash_{\text{UJ}} \{p\}\ \bar{\mathsf{s}}\ \{q\}$ for any premises $\forall\tau \cdot \Gamma_\tau \vdash_{\text{LWJ}} \{p\tau\}\ \bar{\mathsf{s}}\ \{q\tau\}$ of the inference rule in the LWJ proof system.

Case LWJ-implication. Assume that

$$\forall\tau \cdot \Gamma_\tau \vdash_{\text{LWJ}} \{p\tau\}\ \overline{s}\ \{q\tau\}$$

is derived from the premises $\forall\tau \cdot \vdash p\tau \Rightarrow p'\tau$, $\forall\tau \cdot \vdash q'\tau \Rightarrow q\tau$, and $\forall\tau \cdot \Gamma_\tau \vdash_{\text{LWJ}} \{p'\tau\}\ \overline{s}\ \{q'\tau\}$. It follows directly from the IH that

$$\Gamma \vdash_{\text{UJ}} \{p'\}\ \overline{s}\ \{q'\}.$$

From the soundness of first-order logic, we know that $\forall\tau \cdot \models p\tau \Rightarrow p'\tau$ and $\forall\tau \cdot \models q'\tau \Rightarrow q\tau$, and then, by Definition 3.3, $\models p \Rightarrow p'$ and $\models q' \Rightarrow q$. It then follows by relative completeness that $\vdash p \Rightarrow p'$ and $\vdash q' \Rightarrow q$, and by UJ-implication, we derive

$$\Gamma \vdash_{\text{UJ}} \{p\}\ \overline{s}\ \{q\}.$$

Case LWJ-ifElse. Assume

$$\forall\tau \cdot \Gamma_\tau \vdash_{\text{LWJ}} \{p\tau\}\mathbf{if}\ (\mathsf{e})\ \overline{\mathsf{s}_1}\ \mathbf{else}\ \overline{\mathsf{s}_2}\ \mathbf{fi}\{q\tau\}.$$

By rule LWJ-ifElse, we have

$$\forall\tau \cdot \Gamma_\tau \vdash_{\text{LWJ}} \{p\tau \wedge \mathsf{e}\}\overline{\mathsf{s}_1}\{q\tau\}$$

and

$$\forall\tau \cdot \Gamma_\tau \vdash_{\text{LWJ}} \{p\tau \wedge \neg\mathsf{e}\}\overline{\mathsf{s}_2}\{q\tau\}$$

From Definition 3.2, we know that $p\tau \wedge \mathsf{e} = (p \wedge \mathsf{e})\tau$ and $p\tau \wedge \neg\mathsf{e} = (p \wedge \neg\mathsf{e})\tau$, so by the IH we get

$$\Gamma \vdash_{\text{UJ}} \{p \wedge \mathsf{e}\}\overline{\mathsf{s}_1}\{q\}$$

and

$$\Gamma \vdash_{\text{UJ}} \{p \wedge \neg\mathsf{e}\}\overline{\mathsf{s}_2}\{q\}$$

By UJ-ifElse we can then derive

$$\Gamma \vdash_{\text{UJ}} \{p\}\mathbf{if}\ (\mathsf{e})\ \overline{\mathsf{s}_1}\ \mathbf{else}\ \overline{\mathsf{s}_2}\ \mathbf{fi}\{q\}.$$

Case LWJ-composition. Assume

$$\forall\tau \cdot \Gamma_\tau \vdash_{\text{LWJ}} \{p\tau\}\overline{\mathsf{s}_1}; \overline{\mathsf{s}_2}\{q\tau\}.$$

By LWJ-composition, we have

$$\forall\tau \cdot \Gamma_\tau \vdash_{\text{LWJ}} \{p\tau\}\overline{\mathsf{s}_1}\{g\tau\}$$

and

$$\forall \tau \cdot \Gamma_\tau \vdash_{\mathrm{LWJ}} \{g\tau\}\overline{\mathsf{s}_2}\{q\tau\}.$$

By the IH, we get

$$\Gamma \vdash_{\mathrm{UJ}} \{p\}\overline{\mathsf{s}_1}\{g\}$$

and

$$\Gamma \vdash_{\mathrm{UJ}} \{g\}\overline{\mathsf{s}_2}\{q\}.$$

By UJ-composition, we then derive

$$\Gamma \vdash_{\mathrm{UJ}} \{p\}\overline{\mathsf{s}_1}; \overline{\mathsf{s}_2}\{q\}.$$

Case LWJ-internal. Assume

$$\forall \tau \cdot \Gamma_\tau \vdash_{\mathrm{LWJ}} \{p[\overline{\mathsf{y}}, \overline{\mathsf{x}} := \overline{\mathsf{y}'}, \overline{\mathsf{e}}]\tau\}\mathsf{w} = \mathsf{m}(\overline{\mathsf{e}})\{q[\overline{\mathsf{y}}, \overline{\mathsf{x}}, \mathsf{result} := \overline{\mathsf{y}''}, \overline{\mathsf{e}}, \mathsf{w}]\tau\}.$$

By Definition 3.2, this is equivalent to

$$\forall \tau \cdot \Gamma_\tau \vdash_{\mathrm{LWJ}} \{p\tau[\overline{\mathsf{y}}, \overline{\mathsf{x}} := \overline{\mathsf{y}'}, \overline{\mathsf{e}}]\}\mathsf{w} = \mathsf{m}(\overline{\mathsf{e}})\{q\tau[\overline{\mathsf{y}}, \overline{\mathsf{x}}, \mathsf{result} := \overline{\mathsf{y}''}, \overline{\mathsf{e}}, \mathsf{w}]\}.$$

By LWJ-internal we know that

$$\forall \tau \cdot \mathsf{m}(\overline{\mathsf{N}\ \mathsf{x}}) : (p\tau, q\tau) \in \Gamma_\tau$$

and, by Definition 3.2,

$$\mathsf{m}(\overline{\mathsf{N}\ \mathsf{x}}) : (p, q) \in \Gamma.$$

By rule UJ-internal, we can then conclude

$$\Gamma \vdash_{\mathrm{UJ}} \{p[\overline{\mathsf{y}}, \overline{\mathsf{x}} := \overline{\mathsf{y}'}, \overline{\mathsf{e}}]\}\mathsf{w} = \mathsf{m}(\overline{\mathsf{e}})\{q[\overline{\mathsf{y}}, \overline{\mathsf{x}}, \mathsf{result} := \overline{\mathsf{y}''}, \overline{\mathsf{e}}, \mathsf{w}]\}.$$

.

Case LWJ-external. Assume

$$\forall \tau \cdot \Gamma_\tau \vdash_{\mathrm{LWJ}} \{p\tau^{\bar{\mathsf{f}}, \overline{\mathsf{y}}, \mathsf{this}, \overline{\mathsf{x}}}_{\overline{\mathsf{f}'}, \overline{\mathsf{y}'}, \mathsf{v}, \overline{\mathsf{e}}}\}\mathsf{w} = \mathsf{v}.\mathsf{m}(\overline{\mathsf{e}})\{q\tau^{\bar{\mathsf{f}}, \overline{\mathsf{y}}, \mathsf{this}, \overline{\mathsf{x}}, \mathsf{result}}_{\overline{\mathsf{f}''}, \overline{\mathsf{y}''}, \mathsf{v}, \overline{\mathsf{e}}, \mathsf{w}}\tau\}.$$

By Definition 3.2, this is equivalent to

$$\forall \tau \cdot \Gamma_\tau \vdash_{\mathrm{LWJ}} \{p\tau^{\bar{\mathsf{f}}, \overline{\mathsf{y}}, \mathsf{this}, \overline{\mathsf{x}}}_{\overline{\mathsf{f}'}, \overline{\mathsf{y}'}, \mathsf{v}, \overline{\mathsf{e}}}\}\mathsf{w} = \mathsf{v}.\mathsf{m}(\overline{\mathsf{e}})\{q\tau^{\bar{\mathsf{f}}, \overline{\mathsf{y}}, \mathsf{this}, \overline{\mathsf{x}}, \mathsf{result}}_{\overline{\mathsf{f}''}, \overline{\mathsf{y}''}, \mathsf{v}, \overline{\mathsf{e}}, \mathsf{w}}\}.$$

By rule LWJ-external, we know that

$$\mathsf{m}(\overline{\mathsf{N}\ \mathsf{x}}) : (\mathbf{p}\tau, \mathbf{q}\tau) \in \mathbf{\Gamma}_\tau$$

and, by Definition 3.2,

$$\mathsf{m}(\overline{\mathsf{N}\ \mathsf{x}}) : (\mathbf{p}, \mathbf{q}) \in \mathbf{\Gamma}.$$

By rule UJ-internal, we can then conclude

$$\Gamma \vdash_{\mathrm{UJ}} \{p[\bar{\mathsf{f}}, \overline{\mathsf{y}}, \mathsf{this}, \overline{\mathsf{x}} := \overline{\mathsf{f}'}, \overline{\mathsf{y}'}, \mathsf{v}, \overline{\mathsf{e}}]\}\mathsf{w} = \mathsf{v}.\mathsf{m}(\overline{\mathsf{e}})\{q[\bar{\mathsf{f}}, \overline{\mathsf{y}}, \mathsf{this}, \overline{\mathsf{x}}, \mathsf{result} := \overline{\mathsf{f}''}, \overline{\mathsf{y}''}, \mathsf{v}, \overline{\mathsf{e}}, \mathsf{w}]\}.$$

□