Supporting subclassing and traits in Syxc

Master's thesis description

Andres Bühlmann

August 28, 2012

Overview

Today, permission-logics such as separation logic [11] and implicit dynamic frames [14] are the basis of many modular program verifiers. For instance, VeriFast [5] and jStar [4] are program verifiers relying on separation logic, whereas VeriCool [15], Chalice [7] and Syxc [12] are program verifiers relying on implicit dynamic frames. With the development of abstract predicate families [8, 9] in separation logic, support for behavioural subtyping has become feasible and is supported by both VeriFast and jStar. Due to the similarity of separation logic and implicit dynamic frames [10], it was possible to adapt the idea of predicate families to the implicit dynamic frame approach. VeriCool is an example thereof. Until now, Chalice and Syxc have not taken this step and therefore do not support behavioural subtyping.

In addition to the well understood concept of subtyping, the less known concept of traits has gained popularity recently. Traits are in particular promoted by the programming language Scala which is itself gaining more and more attention. While traits have been accepted in the Scala community and are also investigated in the research community, there is no program verifier supporting traits so far. One reason may simply be that traits are not yet used in main stream languages and therefore lack attention. Another reason may be the novel aspects arising from the verification of traits. Most importantly, it is not yet clear how to deal with super-calls in traits which can not always be statically bound due to the unknown linearization order of mixin components. A possible but only theoretical solution has been provided by Schwerhoff [13]. However, how the mentioned solution can be automated using an approach based on implicit dynamic frames and predicate families is still an open question.

The first goal of this master's thesis is to add support for behavioural subtyping to Syxc according to the idea of predicate families as demonstrated for instance by VeriCool. Beside adding this support, it shall also be

investigated how this addition affects existing programming concepts such as for instance monitor invariants. In case issues with other programming concepts are discovered, solutions to the found issues shall be developed and implemented as part of the first goal.

The second goal is to add support for traits. As traits are to some degree similar to subtyping it shall be investigated whether there are new language concepts allowing the verification of traits and at the same time could be useful for subtyping.

These two goals are in line with Syxc's long-term goal of verification support for the Scala programming language.

Chalice

Chalice [7] is the name of an experimental programming language and also of a static program verifier based on verification condition generation.

The programming language Chalice was designed having specification and verification of concurrent programs in mind. It is object based, supports fork-join concurrency, predicates, fine-grained locking, fractional permissions, monitors, pure functions as well as the annotation of pre- and post conditions. However, it currently lacks support for standard and well established programming language features such as subtyping and inheritance.

The program verifier Chalice encodes a given Chalice program in the Boogie [1] programming language. The Boogie verifier then generates verification conditions which get discharged by the theorem prover Z3 [3].

Syxc

Syxc [12] is a program verifier for a subset of the Chalice programming language. In contrast to the verifier Chalice which is based on verification condition generation, Syxc is based on symbolic execution [15].

The motivation to use a symbolic execution approach rather than an approach based on verification condition generation has two sources. Firstly, despite symbolic execution having in theory a worst-case runtime which is exponential in the number of branches, it is claimed that for reasonable examples from the literature symbolic execution is roughly twice as fast as verification condition generation [6]. Secondly, the symbolic execution approach seems to be more programmer-friendly as failing proof obligation are much smaller than in verification condition generation based approaches and therefore can be inspected more easily by a programmer.

Goals of this master's thesis

The goal of this master's thesis is to add support for additional language concepts to Syxc. For each added language concept, the verification support has to be implemented eventually.

The language concepts to be added are given by the following list:

- Abstract base-classes
- Interfaces/Contracts
- Subtyping
- Code/Specification-inheritance
- Traits (in the sense of Scala)

The following code examples serve as a motivation for the goals and extensions that are associated with the examples. Please note that many of Scala's language features, in particular generics and inner classes, are not supported by Syxc at this time. Therefore, the selection of examples is heavily limited by that and we restrict us to design patterns and reasonable code examples used in the literature.

Part 1: Supporting behavioural subtyping

- Cell-family example [4, 9]
  Challenge: Verification of behavioural subtyping
  Tested goals: Behavioural subtyping, code/specification-inheritance
- Set interface/contract and conforming implementations based on a list and a tree respectively
  Challenge: Interfaces are required to specify permissions. But the required permissions depend on the implementation. Thus, some sort of mapping is required.
  Tested goals: Interfaces/Contracts, mapping of permissions
- Subtyping-diamond with interfaces (Person, Student, Employee, EmployeeStudent)
  Challenge: The diamond itself and the mapping of specified permissions in the interface to permissions of the implementation. In particular the case where the interface permission of two interfaces map to the same implementation permission.
  Tested goals: Interfaces, mapping of permissions

Part 2: Supporting traits

- Rich interfaces using traits
  Description: Class `List` providing basic list functionality. Trait `Richlist` provides convenience methods based on methods of class `List`.
  Challenge: Verification of trait usage where the trait inherits from a super-class.
  Tested goals: Traits

- Subject/Observer pattern [4] (trait and abstract base-class version)
  Challenge: Verification of traits where the trait has state, implementation and abstract methods. The interesting question is how to deal with the permissions corresponding to the state of the trait. Furthermore, the observers monitor invariant depends on state of the subject which complicates the handling of the permissions and monitor invariants.
  Tested goals: Abstract base-classes, traits, monitor invariants

- Stackable example [13]
  Challenge: Due to potentially unknown linearization order of mixin components, super-calls can not be statically bound in general and the contracts are therefore unknown at verification time.
  Tested goals: Traits, contracts

Extensions

- Channel example
  Challenge: Can we support behavioural subtyping for channels as well?
  Tested goals: Behavioural subtyping for channels

- Cell-family example (DCell in particular) [4, 9]
  Challenge: Allow non-behavioural subtyping
  Tested goals: Non-behavioural subtyping, code/specification-inheritance

- Concurrent cell-family example
  Description: The concurrent cell-family is an extension of the cell-family example [4, 9] in which the instances are shared and additional interfaces are provided. Thus, a client needs to use the acquire and release statements and can make use of the monitor invariants.
  Challenge: If clients can use acquire and release/share statements on the same instance but with different static type, soundness problems may arise as the monitor invariants of the static types could, if allowed, differ. The challenge is to specify how monitor invariants can be changed in subtypes and if required to redefine the semantics of acquire, release and share statements such that soundness is preserved.
  Tested goals: Behavioural subtyping, monitor invariants

- Visitor pattern [4] (Abstract base class and interface version)
  Challenge: Verification of a double-dispatch scenario, postcondition referencing and chaining of referenced postconditions. With chaining of referenced postconditions a kind of meta postcondition is meant that is able to express, for instance, that a certain method is called twice in succession.
  Tested goals: Subtyping, Interfaces, abstract base classes, condition referencing, postcondition chaining

- Solution to the expression problem
  Challenge: Verification of abstract types and explicitly typed self references, inner classes

Related work

We give a brief summary of automated verifiers based on permission-logics supporting object oriented concepts and in particular inheritance. All of the discussed verifiers VeriCool, VeriFast and jStar support behavioural subtyping and have in common that their approach for handling subtyping is based on abstract predicate families [8, 9].

VeriFast [5] is a program verifier supporting the C and Java [16] programming languages. The program is annotated with the help of separation logic. The concept of abstract predicate families has been incorporated by dynamically bound instance predicates (predicate families with an implicit this- and type-parameter). The verification is based on symbolic execution.

VeriCool [15] is a program verifier supporting the Java programming language. It makes use of implicit dynamic frames and therefore has many similarities to Chalice. Concerning verification, verification condition generation and symbolic execution are supported.

jStar [4] is a program verifier supporting the Java programming language. The program is annotated with the help of separation logic in a similar way to VeriFast. jStar is built on top of coreStar [2] which is a verification framework for separation logic. The verification is based on symbolic execution.

Bibliography

[1] Mike Barnett, Bor yuh Evan Chang, Robert Deline, Bart Jacobs, and K. Rustanm. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005, volume 4111 of Lecture Notes in Computer Science*, pages 364–387. Springer, 2006.

[2] Matko Botinčan, Dino Distefano, Mike Dodds, Radu Grigore, and Matthew J. Parkinson. corestar: The core of jstar. In *In Boogie*, pages 65–77, 2011.

[3] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient smt solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[4] Dino Distefano and Matthew J. Parkinson J. jstar: towards practical verification for java. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA '08, pages 213–226, New York, NY, USA, 2008. ACM.

[5] Bart Jacobs and Frank Piessens. The verifast program verifier, 2008.

[6] Ioannis T. Kassios, Peter Müller, and Malte Schwerhoff. Comparing verification condition generation with symbolic execution: an experience report. In *Proceedings of the 4th international conference on Verified Software: theories, tools, experiments*, VSTTE'12, pages 196–208, Berlin, Heidelberg, 2012. Springer-Verlag.

[7] K. Leino, Peter Müller, and Jan Smans. Verification of concurrent programs with chalice. In Alessandro Aldini, Gilles Barthe, and Roberto Gorrieri, editors, *Foundations of Security Analysis and Design V*, volume 5705 of *Lecture Notes in Computer Science*, pages 195–222. Springer Berlin / Heidelberg, 2009.

[8] Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '05, pages 247–258, New York, NY, USA, 2005. ACM.

[9] Matthew J. Parkinson and Gavin M. Bierman. Separation logic, abstraction and inheritance. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '08, pages 75–86, New York, NY, USA, 2008. ACM.

[10] Matthew J. Parkinson and Alexander J. Summers. The relationship between separation logic and implicit dynamic frames. In *Proceedings of the 20th European conference on Programming languages and systems: part of the joint European conferences on theory and practice of software*, ESOP'11/ETAPS'11, pages 439–458, Berlin, Heidelberg, 2011. Springer-Verlag.

[11] John Reynolds. Separation logic: A logic for shared mutable data structures. pages 55–74. IEEE Computer Society, 2002.

[12] Malte Schwerhoff. Symbolic execution for chalice. Master's thesis, ETH Zürich, 2010.

[13] Malte Schwerhoff. Verifying scala traits. Semester project report, ETH Zürich, 2010.

[14] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 148–172, Berlin, Heidelberg, 2009. Springer-Verlag.

[15] Jan Smans, Bart Jacobs, and Frank Piessens. Symbolic execution for implicit dynamic frames, 2009.

[16] Jan Smans, Bart Jacobs, Frank Piessens, Willem Penninckx, Frédéric Vogels, and Pieter Philippaerts. Verifying java programs with verifast.