Abstract

Reasoning about programs can be tricky and error-prone. Formal verification facilitates the formulation and demonstration of rigorous arguments about program correctness. To be usable and useful, a program logic, i.e. a proof system for program verification, should accommodate the principal mechanisms of the programming language and crystallise the informal reasoning of programmers.

Recently, separation logic emerged as a promising tool for reasoning about shared mutable state, which pervades mainstream programming languages such as C, Java and Eiffel. Parkinson and others proposed a proof system for Object-Oriented (OO) programs that combines separation logic with mechanisms to reason about inheritance and dynamic dispatch. This system can verify a wide range of OO programs and design patterns in a concise way. The flexibility and simplicity of the system made it an attractive target for further improvement and broader application.

The contributions of this thesis address the following problems:

1. Specifying, verifying and using relationships between state abstractions. This is especially important when reasoning about OO programs that use multiple inheritance.

2. Reasoning about executable contracts. Executable contracts are often weak and may perform side-effects. Yet they capture useful design information, are programmer-friendly and assist in debugging.

3. Refinement and correctness by construction. Instead of first writing the code and then proving it correct, these techniques make it possible to write a correct program in the first place. The program and its correctness proof grow and evolve together.

State abstraction mechanisms, such as abstract predicates, are useful for reasoning about programs with modules that encapsulate state and hide information. Parkinson adapted abstract predicates to the OO setting, and

they play a central role in his proof system. Since OO code frequently relies on relationships between state abstractions of a class or a class hierarchy, this thesis enhances Parkinson's system with mechanisms for specifying, verifying and exploiting such relationships. The extension also makes it possible to establish the logical consistency of a class hierarchy without considering implementation details, and it facilitates reasoning about multiple inheritance.

Existing OO code often contains contracts in the form of executable preconditions, postconditions and class invariants. These specifications are typically weaker than separation logic assertions, but they are more lightweight and perhaps more likely to be written by programmers. Contracts also record valuable information about program design and are useful for testing and debugging. This thesis contributes a new technique for using the separation logic assertions to verify that executable contracts will always hold at runtime and that they will not perform unwanted side-effects. As a result, verified contracts need not be monitored at runtime, and they add confidence in the correctness of the code and the separation logic specification.

Correctness by construction is an important feature of a mature engineering discipline. In the context of software engineering, it is realised by a calculus for top-down program development that features refinement as a central technique. A refinement calculus helps to construct correct code from a given specification in a series of steps. This thesis proposes freefinement – an algorithm for obtaining a sound refinement calculus from a modular program logic. The resulting refinement calculus can interoperate closely with the program logic, and it is even possible to reuse and translate proofs between them.

Many aspects of the work also apply to other settings. None of the contributions relies on a specific flavour of separation logic. The work on multiple inheritance subsumes interface inheritance, and the reasoning techniques for executable contracts generalise to non-OO languages that use explicit memory management. Finally, freefinement applies to a great variety of formal systems, including program logics for other languages and type systems.