Automated reasoning for reflective programs

Benjamin Horsfall
Foundations of Software Systems
Department of Informatics
University of Sussex
July 30, 2014

Submitted for the degree of Doctor of Philosophy
University of Sussex.

A little while and I will be gone from among you, whither I cannot tell. From nowhere we come, into nowhere we go. What is life? It is a flash of firefly in the night. It is a breath of a buffalo in the wintertime. It is as the little shadow that runs across the grass and loses itself in the sunset.

Isapo-Muxika

Chief Crowfoot (1825–1890)

UNIVERSITY OF SUSSEX

Benjamin Horsfall, Doctor of Philosophy

Automated reasoning for reflective programs

Summary

Reflective programming allows one to construct programs that manipulate or examine their behaviour or structure at runtime. One of the benefits is the ability to create generic code that is able to adapt to being incorporated into different larger programs, without modifications to suit each concrete setting. Due to the runtime nature of reflection, static verification is difficult and has been largely ignored or only weakly supported. This work focusses on supporting verification for cases where generic code that uses reflection is to be used in a "closed" program where the structure of the program is known in advance.

This thesis first describes extensions to a verification system and semi-automated tool that was developed to reason about heap-manipulating programs which may store executable code on the heap. These extensions enable the tool to support a wider range of programs on account of the ability to provide stronger specifications. The system's underlying logic is an extension of separation logic that includes nested Hoare-triples which describe behaviour of stored code. Using this verification tool, with the crucial enhancements in this work, a specified reflective library has been created.

The resulting work presents an approach where metadata is stored on the heap such that the reflective library can be implemented using primitive commands and then specified and verified, rather than developing new proof rules for the reflective operations. The supported reflective functions characterise a subset of Java's reflection library and the specifications guarantee both memory safety and a degree of functional correctness. To demonstrate the application of the developed solution two case studies are carried out, each of which focuses on different reflection features.

The contribution to knowledge is a first look at how to support semi-automated static verification of reflective programs with meaningful specifications.

Table of Contents

Chapter 1

Introduction

Reflective programming allows a program to inspect and modify its own structure and behaviour at runtime. It is a powerful feature that can enable a program to adapt and evolve during execution. This could be by actually altering the code in the program, or by examining the runtime state of the program and making decisions based on what it finds. One class of reflective programming is the creation of generic code that can be placed into many different programs. By using reflection the generic code can reason about the structure of the concrete program it is set in, and behave accordingly. This has the advantage that the generic code may be modular in the sense that it does not need to be modified to handle concrete programs.

Program verification is a technique that allows code to be formally proved as meeting a specification. Program verification can give stronger assurances than typical black or white box testing, which rely on a comprehensive set of test cases in order to give a strong result. Even then the correct behaviour is not guaranteed. Static verification methods are able to prove, with varying degrees of correctness, that a program will meet its specification before it is ever executed.

Programming with reflection supports runtime reasoning as reflective programs can inspect and manipulate their current state. It may therefore be thought that this runtime nature will mean that attempts at *static* verification is futile, and indeed it has been oft-times ignored or only weakly supported.

Whilst a fully reflective programming language will have the ability to modify code, including loading unknown and untrusted new code, this thesis argues that there is a class of reflective programs that *can* be statically verified. This primarily includes generic code that will later be used in a larger program, but will not modify the code of the program during execution. An example of this is a serialization algorithm which will accept any type of object and use reflection to inspect the appropriate class to create a persistent encoded version of the object with its runtime field values.

Whilst verifying the generic code is useful on its own, it should also be possible to use static verification to ensure that the behaviour of the code is as expected when placed in a concrete program, if the context is "closed" and the entire program's structure is known

in advance.

The key to this form of reflection is having the ability to access the program's structure. This is where *metadata* comes in. Metadata contains a representation *of* the program that can be manipulated *by* the program. In this work the metadata is stored on the heap where it can be manipulated by primitive program commands. This allows the reflective operations to be implemented, specified, and verified, without adding built-in support for reflection in the underlying logic. This has the advantage that the soundness is mostly proved by the existing logic, without developing new proof rules.

The set of reflective operations that are supported are based on those available in Java. The verification takes place in a semi-automated tool, which was originally developed to support a logic for programs with stored code. With a number of extensions, this tool has been adapted to support the verification of reflection.

The following section looks at the background of the principles employed in this thesis to support reasoning about reflection. This begins with separation logic, an extension of Hoare logic that is able to reason about programming languages with a mutable heap. Next, an extension of separation logic is introduced that allows reasoning about programs which can store code on the heap. This is followed by a brief summary of program verification tools that are relevant to the tool that this work has been implemented in. Finally, reflection is discussed in terms of support in current programming languages.

During the course of my research, I collaboratively worked on a research tool, Crowfoot. This tool and its underlying logic is the foundation for this thesis which extends and uses it. The tool is summarized in Section 1.2.

1.1 Background

1.1.1 Separation Logic

Program verification is a useful technique for reasoning about the behaviour of programs. It is particularly important for mission-critical systems that must be robust such that they do not crash, but recent advances have led to more user friendly techniques that can be applied to more everyday programs.

An important development in program verification was Hoare logic [1][2], which is an axiomatic logic where programs can be annotated by a specification, and the axioms and rules of the logic can be used to prove the program meets its specification. Specifications are in the form of a pre-condition and a post-condition, and together with a piece of code they constitute a *Hoare triple*:

$$\{P\}\, C\, \{Q\}$$

This meaning of such a triple is that if starting in a state satisfying the pre-condition P, after executing the code C the state will be in a condition satisfying Q.

Dynamic memory allocation, where the heap can change during a program's execution, is a key feature of several programming languages and allows for data to be shared across functions. The heap contains all the memory that can be managed by the programmer, through commands for memory allocation, deallocation, updating content and reading content. When a command like C's `malloc` is executed, it finds some free memory from the heap and reserves it.

The ability of a programmer to control the memory can also be dangerous and cause programs to fault or behave unexpectedly. The use of program verification here would allow the checking of memory safety, to avoid several errors arising from memory management. Berger and Zorn [3] summarized the types of errors into the following categories:

- **Dangling pointers** — if a pointer doesn't point to a valid location, then the program won't have the expected behaviour. This may happen if x points to a cell, then that cell is deallocated. If a command tries to dereference x, then an error will occur. This error could be because there is no allocated memory at the location, or because that location has since been re-allocated to another process and contains unexpected data.
- **Buffer overflows** — if an attempt to write to the heap goes out-of-bounds, it could overwrite existing data stored in a nearby location.
- **Wild pointers** — similar to dangling pointers, there will be problems if a pointer is not initialized (so it doesn't actually point to any location) at the point of a dereferencing command.
- **Invalid `frees`** — a program may try to free heap cells which have been allocated by another process, causing undefined behaviour. Alternatively, there may be repeated attempts to free the same cell, which can corrupt the heap management system.
- **Memory leaks** — if, once finished, a program doesn't free all of the memory it has reserved during execution, then that memory will be locked-out to other processes, so can never be utilized or freed.

The properties of memory safety can be contrasted with that of functional correctness, which relates input and output values.

In order to reason about programs that use a shared mutable data structure (a heap), *separation logic* was developed [4, 5]. This extension of Hoare logic adds support for heap allocation, access, mutation and deallocation. This reasoning is achieved by adding four key features to assertions in Hoare logic.

$$
\begin{aligned}
\langle \text{assert} \rangle ::= \ & \ldots \\
& | \mathbf{emp} \\
& | \langle \text{exp} \rangle \mapsto \langle \text{exp} \rangle \\
& | \langle \text{assert} \rangle \star \langle \text{assert} \rangle \\
& | \langle \text{assert} \rangle \mathrel{-\!\!*} \langle \text{assert} \rangle
\end{aligned}
$$

Firstly, there is **emp**. This asserts that the heap is empty, where no cells have been allocated. The second formula ($\mapsto$) denotes the state of a singleton heap cell, with the

address on the left-hand side and the contents on the right. For example, $a \mapsto b$ describes a heap cell pointed to by address a, with contents b.

The third is the *separating conjunction* or *spatial conjunction*. This is represented as $(\star)$, and performs a similar function to the standard conjunction $(\wedge)$, except that it describes the structure of the heap. The meaning of the assertion $A \star B$ is that the heap can be divided into disjoint heaplets: the first satisfying the assertion A, and the second satisfying B.

The final formula is the *separating/spatial implication*. The formula $A \mathrel{-\!\!*} B$ states that if the heap were to be extended by a disjoint heap satisfying A, then the new heap would satisfy B.

One of the major benefits of separation logic is the *local reasoning* it brings. By local reasoning it is meant that when giving specifications to a procedure that constitutes a larger program, it is only necessary to specify the portions of the heap that are used by that procedure, without describing the entire heap. It can be assumed that the rest of the heap remains the same. This is allows us to write reasonably sized specification for each procedure. The key to supporting local reasoning in separation logic is the *frame rule*.

In conventional Hoare logic, one can use the rule of constancy [6]. Intuitively, this states that if a command doesn't interfere with a particular assertion, then that assertion will remain intact after execution. A form of this rule is:

$$\textsc{Constancy} \quad \frac{\{P\}\, C\, \{Q\}}{\{P \wedge R\}\, C\, \{Q \wedge R\}} \quad \text{where } fv(R) \text{ not modified by } C$$

This rule allows the local reasoning described above, but it does not work when we have a mutable heap. However, the rule of constancy can be adapted to separation logic using the separating conjunction. This adapted version is known as the *frame rule*:

$$\textsc{Frame} \quad \frac{\{P\}\, C\, \{Q\}}{\{P \star R\}\, C\, \{Q \star R\}} \quad \text{where } fv(R) \text{ not modified by } C$$

This rule allows one to extend the heaplet of P, which is actually used by C, by any other heaplet R whose free variables don't appear in C, and it is assured that R will be preserved after the execution of C. This is due to the spatial conjunction operator, which ensures that any heap cells specified in P can't also be in R (otherwise P and R would not be disjoint).

Separation logic, an extension of Hoare logic, has itself been extended for handling more complex programming features. For instance O'Hearn adapted it for reasoning about concurrent execution [7]. The modularity of separation logic is useful in proving concurrent code where each thread uses disjoint memory. Where two processes have shared memory, O'Hearn uses the ideas of *conditional critical regions* [8]. By defining resource invariants, it becomes possible to assert ownership of heaplets which can pass from process to process.

Another extension looks at supporting the features of object oriented programming, including class inheritance [9].

The next section looks at work on another extension, whose focus is on supporting reasoning for programs where *code* is stored on the heap.

1.1.2 Higher-order store

The original version of separation logic does not cater for languages which allow pointers to code, instead supporting only a simple heap with value types such as integers. Programs that are able to store code on the heap have been termed *higher-order store* programs [10]. This feature allows a program to create new code during execution, which can later be evaluated. This also includes support for dynamic loading where modules may be loaded on demand such as drivers for an operating system [11], or applying in situ updates of a running program [12]. Complexities arise here because higher order heaps give rise to recursion through the store where the stored code may itself invoke more code on the heap. Such situations where recursion may take place through the heap is akin to "Landin's knot" [13].

For a program to use higher-order store it is necessary to have a method for storing unevaluated code onto the heap. In [10], which presented a Hoare logic for higher-order store, the programming language included *quoted code* on the right-hand side of an assignment. This quoted code is regular program code, except surrounded by '...' such that it isn't immediately executed. Another command to later *unquote*, or execute, the code is needed, which was run. Code could then be written such as:

$$x := `n := n + 1;\ x := `n := n + 2'';$$
$$n := 0;\ \mathsf{run}\ x;$$
$$n := 0;\ \mathsf{run}\ x$$

where x contains some code, which initially increments the value of n before updating itself with code that adds two to n. The second run command will therefore have different behaviour to the first. A crucial contribution in [10] was a rule that can reason about the recursion that is possible as a result of having stored code.

The early work on reasoning about higher-order store programs in [10] used a simple imperative programming language with higher-order store to demonstrate some new Hoare-style proof rules, allowing the verification of (possibly recursive) programs with stored code. The work only looked at parameterless procedures, didn't include commands for memory (de-)allocation in the language, and did not take advantage of the modularity of separation logic.

The work was later developed to utilize and extend separation logic's pointer-handling ability in [14]. This research added the missing constructs for dynamic memory allocation and developed new rules for dealing with the recursive and non-recursive eval (or run) calls. Still lacking support for procedures with parameters, it also had the drawbacks of requiring the stored code to be explicitly contained within assertions.

An alternative approach uses *nested Hoare triples* to reason about higher-order programs in PCF [15]. Initially this supported high-order stored functions, but lacked dynamic memory allocation that gives local state. This was addressed later in [16], which supports the ML-like "let ref". In [17], a similar concept of nested triples is adopted, but this time in the separation logic setting. An assertion language allowing nested triples can eliminate the need to explicitly keep the code of a cell in an assertion. The nested triple instead only specifies the *behaviour* of the code. For example, $\{f \mapsto \{P\} \cdot \{Q\}\}\, eval[f]\, \{f \mapsto \{P\} \cdot \{Q\}\}$ describes the behaviour of the code stored at address f, without needing to show the actual code implementation. An advantage of this approach is that it provides an elegant approach to verifying mutual recursion through the store. If a function pointer in an assertion contained only the concrete commands ($\{f \mapsto$ 'C'$\}$), then it is necessary to verify all those commands during the verification of the calling procedure. Simply specifying the behaviour ($\{f \mapsto \{P\} \cdot \{Q\}\}$) means that each mutually recursive procedure can be verified individually (thus we have modular reasoning).

A "deep frame rule" is also introduced in [17] [18]. By defining the invariant extension operator $\otimes$ on assertions, an invariant may be $\star$-conjoined on to the pre- and post-conditions of nested triples. For example the assertion $\{a \mapsto 0\}\, k\, \{a \mapsto 1\} \otimes R$ is the equivalent to $\{a \mapsto 0 \star R\}\, k\, \{a \mapsto 1 \star R\}$. The operator $\circ$ is used as an abbreviation: $P \circ R \Leftrightarrow (P \otimes R) \star R$. The distribution rules for $\otimes$ through the small assertion language used in [17] are given in Figure 1.1.

The deep version of the frame rule can then be defined as follows:

$$\frac{\{P\}\, k\, \{Q\}}{\{P \circ R\}\, k\, \{Q \circ R\}}$$

It is noted in the work of [17] that an axiom version of the deepframe rule ($\{P\}\, k\, \{Q\} \Rightarrow \{P \circ R\}\, k\, \{Q \circ R\}$), which allows the invariant to be selectively added to certain triples, is not sound in their logic [18]. This is in contrast to the standard or shallow frame axiom ($\{P\}\, k\, \{Q\} \Rightarrow \{P \star R\}\, k\, \{Q \star R\}$), which is sound. An observation was made in [19] that shows that, while not sound in general, in certain cases the deep frame axiom can be used safely and suggests that such cases can be given a form of specification that guarantees a promise of being well-behaved.

$$\begin{array}{rcll}
\{P\}\, e\, \{Q\} \otimes R & \Leftrightarrow & \{P \circ R\}\, e\, \{Q \circ R\} & \\
(P \otimes R') & \Leftrightarrow & P \otimes (R' \circ R) & \\
(\exists x.P) \otimes R & \Leftrightarrow & \exists x.(P \otimes R) & \\
(P \oplus Q) \otimes R & \Leftrightarrow & (P \otimes R) \oplus (Q \otimes R) & (\oplus \in \{\Rightarrow, \wedge, \vee, \star\}) \\
P \otimes R & \Leftrightarrow & P & \text{if } P \text{ is: } \mathit{true}, \mathit{false}, \mathsf{emp}, e \mapsto e', \text{ or } e = e'
\end{array}$$

Figure 1.1: Distribution axioms for the invariant extension operator (from [18])

The work in pairing nested triples with separation logic did not support parameters. This was added to the logic in [20], where patterns of specifications were presented for programs that use recursion through the store. The examples in this work were more complex than the demonstrative applications created previously, and make use of recursive and inductive assertions.

Further details of the separation logic for nested Hoare triples are available in Section 1.2, where a verification tool based on the logic is discussed.

1.1.3 Antiframe rule

Following on from the work on reasoning for higher-order store programs, the (higher-order) *anti*frame rule [21] provides a new method of hiding. This is not to be confused with the anti-frame rule used in bi-abduction [22] (see Section 1.1.4.2), where a frame is inferred to be the *missing* portions of state required by a procedure call (in contrast to the ordinary frame being the *left-over* portions).

Part of a program will often maintain some private local state. For instance an object may have private fields that are not visible in it's interface. This is of benefit to programmers because the it keeps the complexity under control, and it makes sense that the same advantage should apply to specifications. Pottier uses an example of a memory manager: when deallocating some memory, the underlying memory manager may maintain a free-list. However this list is of no interest to a program that is calling the deallocation procedure, and ideally would not appear in its specification. The standard and higher-order frame rules, while modular in that portions of the heap that are not used by a callee can be safely added, does not support this other kind of hiding as all portions of the heap that are used must be included in the specification.

The antiframe rule, as the name suggests is the dual of the frame rule. When introduced by Pottier it is in the context of a type-capability system for an ML-like language, [23]. It takes the form:

$$\frac{\Gamma \otimes C \vdash t : (\chi \otimes C) \star C}{\Gamma \vdash t : \chi}$$

for some state described by the capability C. This rule means that if a term t uses some local state C, but for every interaction with the context Γ the capability is maintained (enforced by the invariant extension $\Gamma \otimes C$), then it is safe to assume the type of t *without* the capability C.

The first soundness proof for the antiframe in the type-capability system was published in [24]. This was based on an earlier step-indexing model of the type and capability system whose soundness was proved without the antiframe rule in [25].

An adaptation of the antiframe rule to a separation logic setting was undertaken in [26] (which includes higher-order store, but not higher-order functions). The separation logic form of the rule is:

$$\frac{\{P \otimes R\}\, C\, \{(Q \otimes R) \star R\}}{\{P\}\, C\, \{Q\}}$$

The soundness of the separation logic version of the antiframe rule used denotational semantics and required solving a non-trivial recursive domain equation.

1.1.4 Automated verification

With the development of proof rules for verifying programs, as discussed in the previous section, it can be useful to implement them in a tool providing some automation. This can be helpful due to the time it takes for manual verification, and also the possibility of human error introducing mistakes into the proofs. With large programs and complex specifications, manual verification would be challenging. A verification tool should allow a programmer to input their program source code to the verifier, along with a specification, and get a verification result containing information on any parts that fail.

The level of automation, however, varies from tool to tool. A fully automated verification tool would not need any additional input from the human verifier, except for the input program. Typically however the input program is first annotated, at the very least with specifications. Some of the tools outlined below require extra information such as loop invariants or hints for handling complex assertions.

This section describes some of the most relevant (semi-)automatic verification tools which are based separation logic.

1.1.4.1 Smallfoot

The earliest tool to use separation logic is Smallfoot [27, 28]. This tool accepts simple imperative programs, which have been annotated with pre- and post-conditions along with any loop invariants. Smallfoot proves mostly memory safety and lacks pure formulae in its assertion language to provide further functional correctness. A later paper [29] also mentions the ability to handle the concurrent extensions of separation logic described by O'Hearn [7].

The assertion language used to give the pre- and post-conditions includes some built-in predicates for referring to special data-structures, such as for trees and lists. These are inductively defined and allow assertions such as $ls(x, y)$, representing a list segment from address x ending at address y.

A key concept employed in this tool (and many others) is *symbolic execution*. Symbolic execution makes it possible to see what the behaviour of executing a program is, but at an abstract level without the need to provide concrete input values. The work on symbolic execution in Smallfoot has roots in work on Shape Analysis [30] and the Pointer Assertion Logic Engine (PALE) tool [31], and has since been a basis for many other tools.

Another contribution of Smallfoot are techniques for automated entailment checking. This step will typically occur during procedure calls, and at the end of the symbolic execution when testing whether the symbolic state fulfils the given post-condition. Complexities arise here when the entailment includes one or more of the built-in predicates. For example, if the post-condition is $ls(x, y)$ the verifier needs to know that $x \mapsto 0, y \star y \mapsto 0, 0$ does fulfil it. This is achieved using additional rules which 'roll up' parts of an assertion into an equivalent predicate, which can then be matched with the postcondition. The entailment step includes rules known as *subtraction rules*, which attempt to remove matching parts from each side of an entailment. Eventually, if the post-condition is fulfilled, then the entailment should have been reduced to $\mathsf{emp} \vdash \mathsf{emp}$.

Finally, they devise a technique to perform *frame inference*. This allows the symbolic execution to proceed when the current symbolic state includes more information than is required by the pre-condition of the relevant execution rule. For example, for deallocating a cell, `dispose(x)` will only require a pre-condition of $x \mapsto \_$. But the rule should still fire if the current state looks like $x \mapsto \_ \star y \mapsto \_$. Due to the frame rule from separation logic, we know that the second part will be preserved after execution, so the symbolic execution step of the verifier should infer the left-over frame part (here $y \mapsto \_$).

One of the limitations of the Smallfoot system is that the assertion language is restricted to memory safety. The assertions are also limited to the built-in predicates for linked lists and trees, lacking the possibility of the users to give their own inductive definitions.

1.1.4.2 Space Invader

Space invader [32] is a tool which attempts to "discover the shapes of data structures in the heap". This technique is known as *shape analysis* [30], which has been adapted to separation logic [33].

Abstract interpretation is used to automatically generate post-conditions and loop invariants from a given pre-condition and piece of code. For the invariants, they can be generated by symbolically executing the loop code, and at the end of each loop, try to abstract away the symbolic state using, for instance, a list definition predicate. The idea is that eventually the fixed-point will be reached and that can be the invariant.

Bi-abduction [22] is a later extension to SpaceInvader. This uses compositional ideas in order to verify an entire program, without needing any specification annotations. By organising all procedures into levels, such that those procedures that don't call any other are on one level, and the next level contains procedures that call those on the first level and so on. Then, the procedures of the first level can be analysed individually, starting with the empty heap. Then, with the discovered pre- and post-condition specifications from the lower levels, analyse those procedures on the next level up.

This principle requires the opposite of the frame inference rule (which they call 'abduction'). The frame rule looks at what is *left-over* from an execution, whereas abduction endeavours to discover what is *missing* to successfully perform an execution.

The bi-abduction research has been implemented in the SpaceInvader tool for analysing C-programs. Remarkably, they have been able to analyse large real-world programs such as a Linux kernel and provide feedback on the existence of memory leaks [34].

There are weaknesses to this bi-abduction tool, including the lack of support for arrays and pointer-arithmetic. Additionally, many programs will utilize one of the C libraries. In order fully support library use it is necessary to provide specifications for each one.

1.1.4.3 jStar

jStar [35] is a tool that provides a means for verifying Java programs. Java is a widely-used object-oriented language, and it is these OO features such as inheritance and specialization that make verification difficult. jStar is also based on the entailment checking from Smallfoot.

A key foundation of this work is the use of *abstract predicate families* [36][9][37]. Abstract predicate families provide the means to reason about class hierarchies, such that programs making use of inheritance can be verified in a modular way.

When giving a method specification, it is necessary to provide both a *static specification* and a *dynamic specification*. The static specification is used when a method call takes place and the exact type of the object is known, as with direct method calls. This static specification is used to verify the method. However, the dynamic specification is used when there is dynamic dispatch, and these specifications are typically more abstract.

In a class system, dynamic dispatch is where a method is implemented in multiple classes (say `a` and `b`, who are subclasses of `abstract c`), and the implementation to use at a method call on a target object of type `c` is decided dynamically (at run-time) based on the current type of the target object.

The dynamic specification is used so that when a subclass overrides a method, the new behaviour should still satisfy the super-classes dynamic specification. For this reason, the dynamic specification is typically more abstract than the static specification. The abstract predicates families are predicates which are indexed by class. This index allows individual classes to use different definitions for the same predicate.

The jStar paper gives details of the system through a series of examples. These examples are instances of common design patterns, such as Visitor or Subject/Observer, which cover a broad range of object oriented programming principles.

The jStar framework was later generalized to a generic separation logic verification system, coreStar, aimed at providing a bases to be extended for other tools [38].

1.1.4.4 VeriFast

VeriFast [39][40], like the above tools, uses symbolic execution to verify C-like programs. *requires* and *ensures* contracts are used as the pre and post-conditions, respectively. These assertions are extended to allow the use of predicates, such as list or node definitions. However, unlike Smallfoot, these inductive datatypes are defined by the user, allowing

a greater level of flexibility. In order to fulfil the required entailments, these predicates must be manually (un)rolled by annotating the relevant position with an *open* or *close* instruction for the symbolic execution. Additionally, lemma functions can be written which re-write an assertion so that it is in the form expected by a particular rule. Therefore VeriFast requires more intervention from the user than in Smallfoot, which has built-in normalization and re-arrangement rules because it has only a few built-in inductive definitions to handle.

A later extension of this tool [41] is useful with dynamically-typed (Lisp, Python) / unsafe (C, C++) languages where code is mutable. In statically-typed languages, calls always succeed because they are bound to code satisfying the static type of the call. But with dynamically-typed languages, there is no guarantee that a function pointer points to a valid function throughout the execution. An example of where this is a problem is if we have unloadable modules. Here, if a call is made to a function within a module, there is a chance that the module may have been unloaded at some point in the execution, and so we don't know that the code we are calling still exists.

The idea behind VeriFast is that execution of unloadable code requires abstract permission to read at the address, and 'proof' that the code at the address has the expected behaviour. Unlike abstract predicate families [37], which are indexed by target function, VeriFast uses *parameterized function types*, which allows specifications to talk about generic modules and not explicitly give the module name. VeriFast handles the unloading by allowing module names to appear in annotations, adding the predicate *module(x)* which asserts that x is loaded, and having a module declared as unloadable. To ensure that it is safe to call a library's function, each of the function's pre-conditions should imply *module(this)*. To avoid calling a method through a function pointer to a library which has already been unloaded, the unloadable module must be loaded for the the pre-condition of the function to hold.

VeriFast supports class inheritance by what the authors term *instance predicates*, which are much like abstract predicate families where classes that implement an interface provide their own definitions for an abstract predicate defined in that interface specification.

An advantage of VeriFast is that, similar to SpaceInvader, it uses a C-like language, making it more suitable for industry deployment. However, the number of annotations that are required mean that the user will need to have a good understanding of the proof system. VeriFast was later extended to also verify Java programs [42]. A final notable feature of VeriFast is the integration of an SMT (satisfiability-modulo-theory) solver. They use such a solver to perform pure entailment checks. A detailed description on the use of an SMT-solver to assist with the verification tool is given in [43].

1.1.5 Reflection

Reflection allows a program to inspect and manipulate its own structure and behaviour. It was first introduced by Brian Cantwell Smith [44], whose research implemented reflection

in the LISP programming language. In those early days of reflection, Smith gave the following definition:

> ...the ability of an agent to reason not only introspectively, about its self and internal thought processes, but also externally, about its behaviour and situation in the world.

Smith's definition is quite abstract. A more modern, programming-centred definition is given by Forman and Forman [45]:

> ...the ability of a running program to examine itself and its software environment, and to change what it does depending on what it finds.

The key aspects of reflection are defined as follows:

- **Introspection:** This is essentially the 'read' part of reflection. It is the ability of a program to *inspect* its own structure (like data structures, available functions and variables) and state.
- **Intercession:** Intercession is ability to modify the behaviour or structure of a program, for example by modifying the implementation of a function.
- **Reification:** This is an encoding of the current program state during execution. The state is encoded in some data structure such that it can be manipulated through the programming language.

Reflection has been implemented in a number of languages, taking several different forms. Languages like Common LISP [46] where reflection was first seen interprets functions as lists such that they can be manipulated like any other list. This is a step further than the reflection that is being studied here. The languages most relevant to this thesis are *Java* [47] and *C#* [48].

Java uses the `java.lang.reflect`[1] API for most of its reflection. The important classes are the `Object`, `Class`, `Method`, `Field`, and `Constructor`. Some of the methods in these classes are:

- `Class.getMethods()` — get all the methods of a class. Returns array of `Method` objects.
- `Class.getFields()` — get all the fields of a class. Returns array of `Field` objects.
- `Field.get(obj)` — get the value of `obj`'s field.
- `Method.invoke(obj, args)` — invoke a method, using `obj` as the target object.

An example demonstrating the runtime nature of reflection is given in Figure 1.2. The first line imports the reflection library. The small program includes a single class, Foo, which has an integer field and two methods. One of the methods increments the field, and the other decrements it. The reflection takes place in the main method, which will make use of the command line arguments. The first line of main creates a new Foo object, with

[1] `java.lang.reflect` in Java Platform SE 6 http://docs.oracle.com/javase/6/docs/api/java/lang/reflect/package-summary.html

```
import java.lang.reflect.*;
class Foo {
  int x;
  public Foo (int x) { this.x=x; }
  public void incr() { x++; }
  public void decr() { x--; }

  public static void main(String args[]) {
    Foo obj = new Foo(1);
    try {
      Method m = obj.getClass().getMethod(args[0]);
      m.invoke(obj);
      System.out.println(obj.x);
    } catch (NoSuchMethodException e) {
      // ...
    } catch (IllegalAccessException e) {
      // ...
    } catch (InvocationTargetException e) {
      // ...
    }
  }
}
```

Figure 1.2: A simple example of reflection in Java

the field initialised to 1. Next, inside the try clause, a method is looked up inside the class of our object. The method is then executed, and the value of the object's field is printed.

The name of the method to run is given by the user and so it is not clear statically at compile-time what the behaviour of this program will be. There are three options:

1. The user passes "incr" as the argument, in which case the program will print "2".
2. The user passes "decr", in which case the program will print "0".
3. The user passes another string or no string.

In the third case, where a non-existing method is being sought, Java will throw an exception. This is why the reflection must be surrounded by the try-catch, so that runtime faults can be handled gracefully.

The object-oriented language C# supports reflection much in the same way as Java, with mostly the same selection of reflection methods available. Much of the reflection capability is held in the System.Reflection namespace[2].

Reflection is a powerful feature for a programming language, allowing programs to adapt and evolve during execution, however it has a couple of disadvantages. Firstly, it is difficult to implement a language with support for reflection. Herzeel [49] gives a lengthy discussion on the necessary additions to a standard language, which includes the implementation of

[2]System.Reflection Namespace http://msdn.microsoft.com/en-us/library/system.reflection(v=vs.110).aspx

the data-structure storing the current program structure/state, along with data types and procedures for using it.

Another disadvantage is the performance. Programs that use reflection are typically slower than those that do not because extra processing is required to dynamically obtain the required information on the program's structure. Palsberg and Jay [50] compare a static implementation of the Visitor design pattern (1.17sec) with a reflective implementation performing the same function (5min). This is a very large penalty on performance, though the tests were done nearly 15 years ago. Modern processing power and improvements to the language implementation mean that the time will be significantly reduced.

1.2 The Crowfoot verification tool

To support the project exploring reasoning for programs with a higher-order store, a semi-automatic verification tool was developed to support reasoning about programs using a specification language with nested triples [17]. Using the technique of symbolic execution pioneered by Smallfoot, Crowfoot takes annotated programs and proves both memory safety and functional safety. An online version is available [51].

The work in this thesis uses and extends the Crowfoot verification system. Full details of the prover can be found in [52]. Here, only the key details are provided that are critical to the content in later chapters, along with extensions that were developed for the purposes of the work described in this thesis. Note that this was collaborative work with Bernhard Reus and Nathaniel Charlton.

As the name suggests, Crowfoot's architecture is based on the symbolic execution principles of Smallfoot [28], although it was built from scratch. It is a research prototype that uses its own programming language, which is an imperative procedural language with call-by-value parameter passing. It also supports dynamic memory with pointer arithmetic and includes higher-order store commands for loading code on to the heap and evaluating stored code. The programming language includes only the type of integers.

A simple example program written in the Crowfoot language, with annotations, is given in Figure 1.3. The language includes user-defined recursive predicate definitions, and procedures with pre- and post-condition annotations. The focus of the program is the procedure $list\_map$, which is a higher-order map for a linked list where the function that transforms each element is stored on the heap.

In the example, a simple linked list predicate is defined. Predicate names are preceded by a \$-sign, and they accept a sequence of integer arguments and of set arguments, with the two types segregated by a semi-colon. The predicate \$List$(a;\%L)$ describes a linked list started at address a with contents described in the set $\%L$. The set is made up of pairs, which includes the data (val) as well as the pointer.

The specification of $list\_map$ requires a linked list at address l and, in a separate portion of the heap, a function with behaviour described by the nested Hoare-triple. The nested specification simply states that the function maintains a heap cell at the address of its

```
recdef $List(a; %L) :=
   a = 0 ⋆ %L = ∅
 ∨
   ∃ val, next, %rest. a ↦ val, next ⋆ $List(next; %rest) ⋆ %L = {(a, val)} ∪ %rest;

proc list_map(l, f)
∀ %L.
  pre : $List(l; %L) ⋆ f ↦ ∀ x, r.{r ↦ _}_(x, r){r ↦ _};
  post : ∃ %newL. $List(l; %newL) ⋆ f ↦ ∀ x, r. {r ↦ _}_(x, r){r ↦ _};
{
   locals cur, next;
   ghost 'unfold $List(l; %L)';
   if l = 0 then {
      skip
   } else {
      next := [l + 1];
      call list_map(next, f);
      cur := [l];
      eval [f](cur, l)
   };
   ghost 'fold $List(l; ?)'
}
```

```
proc main(l)
  pre : ∃ %L. $List(l; %L);
  post : ∃ %L. $List(l; %L);
{
   locals fun;
   fun := new 0;
   [fun] := incr(_, _);
   call list_map(l, fun);
   [fun] := add_ten(_, _);
   call list_map(l, fun);
   dispose fun
}
```

```
proc incr(x, res)
  pre : res ↦ _;
  post : res ↦ x + 1;
{
   [res] := x + 1
}

proc add_ten(x, res)
  pre : res ↦ _;
  post : res ↦ x + 10;
{
   [res] := x + 10
}
```

Figure 1.3: Example of a higher-order store program that can be verified by Crowfoot

second argument r. The post-condition of *list_map* states that there is still a linked list and a function on the heap, except that the content of the list may have changed have changed.

The body of the map procedure inductively traverses the linked list, using an address offset $(l + 1)$ to access the pointer to the next element. Dereferencing of addresses in achieved by the square-brackets. The eval command is used to execute the stored code pointed to by f, to which it passes the current element's value (dereferenced on the previous line), and the current element's pointer. As a map function, the intention of f is that the input in the first argument is transformed and the result stored in the cell pointed to by the second argument. The body is annotated with two "ghost" hints. These inform the verification tool that it is necessary to unfold or fold the definition of a predicate. In this example this is to expose the the value of l, which is either 0 in the empty-list case, or it is a pointer.

The procedure *main* takes a list l and applies the map procedure twice, with a different transformation function each time. The Crowfoot language does not include nested procedure constructs like λ, and instead stored code is created by copying fixed procedures. To provide the transformation function, a new heap cell is allocated (with new) and the address stored in variable *fun*. Next, the procedure *incr* is loaded into this cell. The underscores in the arguments leave the parameters in place for the stored procedure – it is also possible to use partial application by using a variable in place of an underscore, in which case the stored code is "fixed" for that value. The map procedure is then called, and the same map procedure is used a second time but with a different transformation operation. Finally the cell storing the function is deallocated with the dispose keyword.

Note that Crowfoot is a procedural language, and has no return values and parameters are immutable. In order to overcome this restriction where a result is produced, the convention is for the last argument (*res* in the case of *incr* and *add_ten*) to be a pointer to a cell, in which the result is stored. This is approach can be seen in the implementation of the two transformation procedures.

1.2.1 Language

The syntax for the programming language is given in Figure 1.4. A distinction is made between address and ordinary value expressions, where addresses are restricted to simple subset to aid reasoning about pointer arithmetic. The shaded parts of atomic statements are annotations and serve as hints to the automated prover. While-loops must be annotated with an invariant. The annotation inst-hints provides a hint as to how to instantiate existential variables that occur on the right hand side of an entailment. These are pairs: the hints '$a = 1, b = 2$' might be an annotation for a procedure call that gives rise to the entailment

$$\%A = \{1\} \cup \{2\} \vdash \exists a, b.\ a \in \%A \star b \in \%A \star a \neq b$$

integer variables x, fixed procedure names $\mathcal{F}$, integer literals n, declared constants c

$$
\begin{array}{rcl}
\text{address expr } e_A & ::= & x \mid c \mid x + n \mid x + c \\
\text{value expr } e_V & ::= & n \mid x \mid c \mid e_V + e_V \mid e_V - e_V \mid e_V \times e_V \\
\text{statement } C & ::= & \mathsf{skip} \mid At \mid C; C \mid \mathsf{if}\ e_V[= \mid \neq \mid < \mid \leq]e_V\ \mathsf{then}\ C\ \mathsf{else}\ C \\
 & & \mid \mathsf{while}\ e_V[= \mid \neq \mid < \mid \leq]e_V\ \ P\ \ \mathsf{do}\ C \\
\text{argument } t & ::= & x \mid c \\
\text{atomic statement } At & ::= & x := e_V \mid x := [e_A] \mid [e_A] := e_V \mid [e_A] := [e_A] \\
 & & \mid x := \mathsf{new}\ {e_V}^+ \mid \mathsf{dispose}\ e_A \mid \mathsf{call}\ \mathcal{F}(t^*)\ \ \text{inst-hints}^*\ \text{deepframe?} \\
 & & \mid \mathsf{eval}\ [e_A](t^*)\ \ \text{inst-hints}^*\ \mid [e_A] := \mathcal{F}([t|\_]^*)\ \ \text{deepframe?} \\
 & & \mid \ \mathsf{ghost}\ G \\
\text{inst-hints} & ::= & x = e_V \mid \alpha = e_S \\
\text{deepframe} & ::= & \mathsf{deepframe}\ \Psi \\
\text{ghost statement } G & ::= & \mathsf{fold}\ \mathrm{P}((x|?)^*; (\alpha|?)^*)\ \text{inst-hints?} \\
 & & \mid \mathsf{unfold}\ \mathrm{P}((x|?)^*; (\alpha|?)^*) \\
 & & \mid \mathsf{split}\ \mathrm{P}\ x\ ((e_V|?)^+) \mid \mathsf{join}\ \mathrm{P}\ x
\end{array}
$$

Figure 1.4: Crowfoot: abstract syntax for program statements.

set variables α, predicate names P

$$
\begin{array}{lrcl}
\text{element expressions} & e_E & ::= & e_V \mid ({e_E}^+) \\
\text{set expressions} & e_S & ::= & \alpha \mid e_S \cup e_S \mid e_S \cap e_S \mid e_S \setminus e_S \mid \{e_E\} \mid \mathrm{proj}_n(e_S) \mid \emptyset \\
\text{behavioural spec.} & B & ::= & \forall [x|\alpha]^*.\ \{P\} \cdot (t^*)\{Q\} \\
\text{content spec.} & \mathscr{C} & ::= & e_V \mid \_ \mid B \\
\text{atomic formula} & A & ::= & e_A \mapsto \mathscr{C}^+ \\
 & & & \mid P({e_V}^*, {e_S}^*) \mid e_V = e_V \mid e_V \neq e_V \\
 & & & \mid e_E \in e_S \mid e_E \notin e_S \mid e_S \subseteq e_S \mid e_S = e_S \\
\text{spatial conjunction} & \Phi, \Theta, \Upsilon & ::= & A \star \Theta \mid \mathsf{emp} \\
\text{assertion disjunction} & \Psi & ::= & \exists [x|\alpha]^*.\Theta \\
\text{assertion} & P, Q & ::= & \mathit{false} \mid \Psi \ \vee \ P
\end{array}
$$

Figure 1.5: Crowfoot: Abstract syntax for the assertion language

when checking the current state fulfils the callee's pre-condition. The "deepframe" annotation can be used to add an invariant to a triple using the $\otimes$ operator. When used in the context of stored-code, the deepframe annotation may be applied at the store-code statement only, and not also at the eval stage. This is important because to add an invariant during an eval would require an axiom version of the deepframe rule, where the invariant is framed selectively to just one nested Hoare-triple, which is unsound [18].

Finally, the ghost statements are used to guide the folding and unfolding of predicate definitions. The arguments to predicates in these hints may be question-marks, in which case the prover will try to guess a suitable argument that fulfils the definition. The split and join cases provide special handling for linked list segments, which is explained below.

Crowfoot's assertion language is in Figure 1.5. The set expressions include a projection

const c; or const $c = n$;	// Constant variables
forall [pure] P($\vec{\_}$; $\vec{\_}$).	// Abstract predicates (with no definition)
recdef P($\vec{x}$; $\vec{\alpha}$) := Q	// Predicate definitions (possibly recursive)
recdef R($\vec{x}$; $\vec{\alpha}$) := $S(\vec{y}) \circ \Psi$	// Predicates using with invariant extension (see below)
proc $\mathcal{F}(\vec{x})$ $\forall \vec{y}, \vec{\alpha}$. pre : P; post : Q; {locals$\vec{v}$; C}	// Procedure declaration
proc abstract $\mathcal{F}(\vec{x})$ $\forall \vec{y}, \vec{\alpha}$. pre : P; post : Q;	// Abstract procedure declaration

Figure 1.6: Abstract syntax for declarations in the Crowfoot language

function, from sets of tuples to sets:

$$\text{proj}_2(\{(a,b,c)\} \cup \{(d,e,f)\} \cup \{(g,h,i)\}) = \{b\} \cup \{e\} \cup \{h\}$$

In the concrete syntax, predicate variables can be identified by a preceding dollar-sign (e.g. \$P()), and set variables by a preceding percent-sign (e.g. $\%S$). There are two commonly used abbreviations for $\mapsto$ formulae:

$$\begin{aligned} a \mapsto x, y, z \quad &\Leftrightarrow \quad a \mapsto x \star a+1 \mapsto y \star a+2 \mapsto z \\ a \mapsto \_ \quad &\Leftrightarrow \quad \exists v.\, a \mapsto v \end{aligned}$$

The input file to Crowfoot is a sequence of declarations, defining predicates and specified procedures. The forms of declarations are given in Figure 1.6.

Predicates that are declared with forall do not have a definition, and so verification will show that the program is correct for any definition. Predicates can be declared using the $\circ$ operator to deeply frame an invariant onto the definition of another predicate.

Crowfoot lets the user define their own predicates, which provides a high-level of flexibility. Additionally however, a special pattern of linked list segments is supported, and built-in rules are able to perform splitting and joining of segments. The recognised list segment definitions must be of the following form, where the shaded parts are instantiated for each concrete definition:

$$\mathsf{recdef}\ \mathrm{L}(s,t,\alpha) \quad := \quad s = t \star \alpha = \emptyset \quad \vee \quad \left(\begin{array}{l} \exists n, \vec{v}, \beta\ . \\ \quad s \mapsto \mathscr{C}_1, \ldots, \mathscr{C}_M\ , n \\ \quad \star\ A_1 \star \cdots \star A_N \\ \quad \star\, \mathrm{P}(n,t,\beta) \\ \quad \star\, \alpha = \{(E_1, \ldots, E_k)\} \cup \beta \end{array} \right)$$

Instances of this pattern of list-segment may be split or joined using the special ghost statements, providing each element can be uniquely identified. Splitting requires knowing that there is some element in the list (e.g. $(E_1, \ldots, E_k) \in \alpha$), and results in three parts: the sought element, a list segment containing all the preceding elements, and a list segment containing all succeeding elements. These splittable list segments are used extensively later in Chapter 3, where metadata is stored on the heap in such linked list structures.

1.2.2 System overview

The crucial parts of the Crowfoot verifier are the symbolic execution engine and the entailment prover. To support the nested Hoare triples, there are actually two parts to the entailment prover: the first for entailments between assertions, and the second for entailments between triples. Because the triples are composed of assertions in the pre- and post-conditions, the two entailment provers must be mutually recursive.

The implementation uses five judgements for verification. The most important for understanding the work in this thesis are:

- $\Phi \vdash^{I} \exists \vec{v}.\Upsilon \star \Theta$: Entailments between assertions. This judgement states that Φ entails $\exists \vec{v}.\Upsilon$, and the left-over frame is Θ. The I is a mapping of the existential variables to discovered instantiations.
- $\Phi \vdash_{SMT} \Phi'$: Entailments between pure assertions, delegated to an SMT solver.
- $B \vdash_{\text{find-post}} \{\Phi\} \cdot (\vec{t})\, \{Q\}$. For procedure calls/evals, computation of a post-condition Q that is the result of running code specified by B in starting state Φ.
- $\Pi; \Gamma \triangleright \{P\}\, C\, \{Q\}$: Symbolic execution, where the validity of the triple depends on a predicate context Π containing user-defined predicate definitions, and procedure context Γ containing declared fixed procedures.

For the entailment proving judgement $\vdash^{I}$, a selection of some of the rules are given in Figure 1.7. These are cancellation rules that match spatial formula on the left and right-hand side and subtract those formula. The middle rule, CancelPtInstContents, also performs an instantiation of the existentially quantified variable v.

An example of one of the symbolic execution rules is given in Figure 1.8. This rule updates the contents of a heap cell. The symbolic execution rules use a continuation style where the command is followed by a continuation C. Note that if existential variables are introduced after symbolically executing a command, they are skolemized by some globally fresh variable. This means that there is never existential quantification in the symbolic state, and consequently on the left-hand side of an entailment.

$$\textsc{CancelPt1}\quad \frac{\Phi \;\vdash^{I}\; \exists \vec{v}\,.\,\Upsilon \star \Theta}{\Phi \star e_A \mapsto \mathscr{C} \;\vdash^{I}\; \exists \vec{v}\,.\,\Upsilon \star e' \mapsto \_ \star \Theta} \quad \begin{array}{l} fv(e') \cap \vec{v} = \emptyset, \\ purify(\Phi) \vdash_{SMT} e_A = e' \end{array}$$

$$\textsc{CancelPtInstContents}\quad \frac{\Phi \;\vdash^{I}\; \exists \vec{v}\,.\,\Upsilon[v \backslash E] \star \Theta}{\Phi \star e_A \mapsto E \;\vdash^{I[v:=E]}\; \exists \vec{v}, v\,.\,\Upsilon \star e' \mapsto v \star \Theta} \quad \begin{array}{l} fv(e') \cap \vec{v}, v = \emptyset, \\ purify(\Phi) \vdash_{SMT} e_A = e' \end{array}$$

$$\textsc{CancelPtTriple}\quad \frac{\Phi \;\vdash^{I}\; \exists \vec{v}\,.\,\Upsilon \star \Theta \qquad B_1 \vdash B_2}{\Phi \star e_A \mapsto B_1 \;\vdash^{I}\; \exists \vec{v}\,.\,\Upsilon \star e' \mapsto B_2 \star \Theta} \quad \begin{array}{l} fv(e', B_2) \cap \vec{v} = \emptyset, \\ purify(\Phi) \vdash_{SMT} e_A = e' \end{array}$$

Figure 1.7: Examples of rules implemented in Crowfoot's entailment prover

$$\textsc{Mutate}\quad \frac{purify(\Upsilon) \vdash_{SMT} E_A = (e'_A + o) \qquad \Pi; \Gamma \triangleright \left\{\Upsilon \star e'_A \mapsto \mathscr{C}_0, \ldots, \mathscr{C}_{o-1}, x, \mathscr{C}_{o+1} \ldots, \mathscr{C}_n\right\} C \left\{Q\right\}}{\Pi; \Gamma \triangleright \left\{\Upsilon \star e'_A \mapsto \mathscr{C}_0, \ldots, \mathscr{C}_n\right\} [E_A] := x;\; C \left\{Q\right\}}$$

Figure 1.8: Examples of Crowfoot's symbolic execution rules

1.2.3 Soundness overview

The soundness of Crowfoot's proof rules is based on the logic in [25], and adapted to the new language. This model uses operational semantics and step-indexing. Again, the full details of the Crowfoot logic are in [52] and it is only very briefly sketched here.

1.2.3.1 Operational semantics

The operational semantics use a configuration (C, s, h), of a command C, environment stack s, and heap h. For example:

$$(x := e_V,\, s \cdot \eta,\, h) \quad \rightsquigarrow \quad (\mathsf{skip},\, s \cdot \eta[x \mapsto [\![e_V]\!]_\eta],\, h)$$

where $s \cdot \eta$ is a stack with topmost environment η. Disjoint heaps are denoted by $h_1 \cdot h_2$.

Heaps are a partial map from address to integers:

$$Heap = \mathbb{N}_{>0} \rightharpoonup \mathbb{Z}$$

To support the "higher-order store" where code can be stored on the heap, an encoding to integer $\lceil \_ \rceil$ is used.

There are semantic judgements for the validity of the predicate and procedure contexts with respect to concrete environments π and γ:

$$\pi \models \Pi \qquad \text{and} \qquad \gamma \models \Gamma$$

1.2.3.2 Semantics of assertions

To start, step indexed predicates are required. Let $UPred(H)$ (for any H) be the set of subsets of $\mathbb{N} \times H$ that are downwards closed in the index part (first component):

$$\{p \subseteq \mathbb{N} \times H \mid \forall (k,h) \in p. \forall j \leq k. (j,h) \in p\}.$$

For the Crowfoot logic, the type of H needs to be

$$H = Env \times SetEnv \times Heap$$

where *Env* is an environment for integer variables and *SetEnv* for set variables.

The uniform predicate $UPred(H)$ is a complete, 1-bounded ultrametric space (CBUltne). There is a $W \in \mathsf{CBUltne}$ satisfying

$$W \cong \tfrac{1}{2}(W \to UPred(H))\,. \tag{1.1}$$

Assertions are modelled as elements of *Pred* which is defined as

$$Pred = \tfrac{1}{2}(W \to UPred(H))$$

and denote the isomorphism $i : Pred \to W$.

Functions are defined for $\otimes : Pred \times W \to Pred$, and $\circ : W \times W \to W$

$$p \otimes w = \lambda w'.\, p(w \circ w')$$

$$w_1 \circ w_2 = i((i^{-1}(w_1) \otimes w_2) \cdot i^{-1}(w_2))\,.$$

1.2.3.3 Semantics of triples

Semantic versions of Hoare triples are defined as follows. Note that $\rightsquigarrow_k^\gamma$ is a k-step reduction relation in the operational semantics which depends on the procedure environment. There is a set of configurations Safe_n^γ, which are those configurations that do not reduce to the special aborting configuration abort (denoting runtime errors including memory faults) in n steps or less.

Definition 1. Let $p, q \in Pred$, $w \in W$, $\eta \in Env$, $\sigma \in SetEnv$, let C be a program statement and let γ be a procedure environment. We define that $w, \eta, \sigma \models_n^\gamma (p, C, q)$ holds iff the following holds: for all $r \in UPred(H)$, all $m < n$, all heaps h, all stacks s, if $(m, \eta, \sigma, h) \in p(w) \star i^{-1}(w)(\text{emp}) \star r$, then:

1. $(C, s \cdot \eta, h) \in \text{Safe}_m^\gamma$.
2. For all $k \leq m$ and all $h' \in Heap$, $\eta' \in Env$, if $(C, s \cdot \eta, h) \rightsquigarrow_k^\gamma (\mathsf{skip}, s \cdot \eta', h')$, then $(m - k, \eta', \sigma, h') \in q(w) \star i^{-1}(w)(\text{emp}) \star r$.

We write $n \models_\pi^\gamma (P, C, Q)$ iff for all $w \in W$ and for all set environments σ and integer environments η it holds that $w, \eta, \sigma \models_n^\gamma (\llbracket P \rrbracket_\pi, C, \llbracket Q \rrbracket_\pi)$. Accordingly we write $\models_\pi^\gamma (P, C, Q)$ for $\forall n \in \mathbb{N}.\, n \models_\pi^\gamma (P, C, Q)$.

$$w_1 \circ w_2 = i\big((i^{-1}(w_1) \otimes w_2) \cdot i^{-1}(w_2)\big) \tag{1.2}$$

1.3 Previous work

Due to the runtime nature of reflection, supporting static verification of reflective programs has largely been overlooked. The form of reflection addressed in this thesis is focussed on the Java-like reflection, mostly contained within the java.lang.reflect API. Those verification systems that verify Java programs, including [34], have mostly not considered the reflection API.

The Java Modelling Language (JML) [53] provides a means for annotating Java programs with specifications. Tools such as ESC/Java2 (Extended Static Checking) tool [54] provides automated checking of the specifications. Specifications have been produced for Java's APIs, however those of the reflection library are mostly incomplete. Their specifications mostly describe the behaviour as "pure", which means they have no side-effects.

Object: The specification for Object.getClass() uses an auxiliary variable named *_getClass* which is assigned to the result of the special type-of operator, which returns the dynamic class of the given object. Additionally the specification states that the return value is not null. This specification is essentially the same as that in this thesis.

```
//@ public model non_null Class _getClass;
//@ public represents _getClass <- \typeof(this);

/*@ public normal_behavior
  @   ensures \result == _getClass;
  @   ensures_redundantly \result != null;
  @*/
public /*@ pure @*/ final /*@non_null*/ Class getClass();
```

Class: The method *Class.getName*(), which returns a string, is specified in JML as returning a string that corresponds to an auxiliary variable. This variable is assigned the value returned by the *getName*() method itself, and this ensures that the result is always the same. However there is no assertion as to what the value of string returned is.

The other methods in Class that are tackled in this thesis are declared simply as pure, or as not returning null, which does not provide much detail of their behaviour.

Constructor: The Constructor methods are all annotated as pure, except for *newInstance*, which is asserted to return an object that is not null.

Field: All of the specifications of Field members are pure, except for the mutator (set) methods, which are unspecified.

Method: Again, the specifications are mostly pure and/or not null. That is except for the *invoke* method. This method's specification describes the behaviour of invoking a method that returns a primitive type, where the result of the reflective invocation is wrapped up into the appropriate wrapper class. It makes no assertion about the behaviour of the underlying method being interfaced with.

1.4 Contributions & content

The content of this thesis is as follows.

Chapter 2 gives details of a number of extensions to the Crowfoot verification system. These extensions are cruicial to the work in the later chapters on verification for reflection, however they are also useful extensions in general, outside the reflection context.

Contribution: The Crowfoot language is enhanced with a string type, sets-of-sets, and a mechanism for applying lemmas during verification. These extensions allow stronger specifications and successful verification of more programs that the system was otherwise unable to prove. Additionally, this chapter includes the first tool support for Pottier's antiframe rule [21].

Chapter 3 presents a library of reflective operations, based on a subset of those in Java's API. The library is implemented in the Crowfoot language with an assumption that Java programs can be systematically translated. The metadata is stored on the heap such that the implementations of the library may use primitive heap manipulation commands. The library procedures have been specified and verified in the tool. Additionally the metadata is generated automatically from a given input program.

Contribution: The specified library is the first attempt at giving strong behavioural specifications to Java's reflective methods, albeit in a different language. These specifications make assertions over the metadata that enable verification to ensure that "exceptions" such as no-such-method will be avoided. The specifications include support for functional correctness of reflectively invoked methods.

Chapter 4 makes use of the reflective library in the previous chapter and undertakes two case studies. The two case studies constitute generic algorithms which are suitable for reuse in a number of programs. The first case study verifies a reflective version of the visitor pattern, where a generic "visit" method can handle any type of object. The second case study verifies algorithms for serialization and deserialization of objects, where objects are encoded in an XML-like structure.

Contribution: The verification case studies validate and demonstrate the application of the library developed in the previous chapter. The examples are not contrived, and are taken from Java developer literature. Together the case studies cover a wide range of reflective operations and show that there are cases when reflective programs can be verified.

Chapter 5 makes use of Pottier's antiframe rule to devise an alternative approach to making a reflective library available to a program. Due to the metadata only being directly accessed by the library, it is demonstrated how it can be hidden from clients. This is achieved by making the metadata a local invariant to the library. In order to accomplish this and apply the antiframe rule, the reflective library is moved to the heap.

Contribution: This work represents a novel application of the antiframe rule at the same time as making the reflective library more intuitive because the metadata, which is redundant to clients in the sense of needing direct access, is no longer visible. This more

closely models Java, where the metadata on the heap is not directly accessible.

Chapter 6 concludes the thesis, summarising the work that has been accomplished. This chapter also looks to the future, considering ways in which this approach to reflection verification can be improved and extended.

1.5 Published work

During the course of my research, which involved collaborative work developing the Crowfoot tool, three co-authored papers have been published, one of which as lead author. Additionally a journal article has been submitted.

- N. Charlton, **B. Horsfall**, and B. Reus. Formal reasoning about runtime code update. In *proceedings of the 2011 IEEE 27th International Conference on Data Engineering* (HotSWUp), pages 134-138, 2011.
- N. Charlton, **B. Horsfall**, and B. Reus. Crowfoot: a verifier for higher-order store programs. In *Verification, Model Checking, and Abstract Interpretation* (VMCAI), pages 136-151, 2012.
- **B. Horsfall**, N. Charlton, and B. Reus. Verifying the reflective visitor pattern. In *proceedings of the 14th Workshop on Formal Techniques for Java-like Programs* (FTfJP), pages 27-34, 2012.
- B. Reus, N. Charlton, and **B. Horsfall**. Symbolic execution proofs for higher order store programs. Submitted to *Journal of Automated Reasoning*, 2013.

Chapter 2

Enhancements to the verification tool

An overview of the Crowfoot system was detailed in Chapter 1. The language and supporting logic was engineered to allow verification of a range of programs utilising a higher-order store, however whilst it was able to handle all the initial example programs it was still limited in both automation and flexibility. For instance there is only the integer type. As the usefulness of the tool for verification of reflective programs became apparent (see later Chapter 3) it became desirable to provide for an enriched language. For example the lack of further types could be addressed by the addition of strings.

This chapter details extensions that were implemented with a primary purpose of supporting the reflection work from the following chapters, however they are general extensions that are justifiably useful in other applications that have no direct relation to the work on reflection.

In summary, the extensions specific to this thesis are:

1. Provable lemmas
2. String variables with (in-)equality and concatenation operator
3. Sets-of-sets in assertions
4. "Anti-frame" rule support
5. A concept of "pure lists", as opposed to heap-based data-structures

These extensions are presented in the following sections, followed by an evaluation and full recap of the Crowfoot language in Section 2.8.

2.1 Lemmas

In the introduction to the Crowfoot verification system in Chapter 1, it was explained that the reasoning for pure entailments is delegated to an SMT solver. This approach does, however, have its limitations. There are cases where the SMT solver is unable to provide a positive answer when one is expected. This can be because it reaches a timeout, or it simply lacks the necessary axioms in its theories to prove un-satisfiability. The limitation

is particularly evident in cases where a number of quantifiers appear, which are common with the encoding of set properties.

In example programs with complicated (large) specifications, the timeout is quite often reached before an answer has been found. Obviously this can be resolved by increasing the timeout, however this can have a disastrous effect on overall performance because the extra time will also be spent on entailments that are expected to fail. A normal case where the entailment is expected to fail is during the search for instantiations for existentially quantified variables, where the first instantiation choice may not be the correct one. From the webpages of the Z3 SMT solver [55], a caution is issued regarding running times:

> **Why can small modifications in the input formula produce significant runtime differences?**
>
> Z3 can be used to solve many intractable problems (from NP-complete to undecidable). So, it is heavily based on heuristics. For example, let us consider the satisfiability of propositional formulas. You can pass propositional formulas where Z3 can immediately identify a satisfying assignment because the search happened to initially choose the right assignments, or Z3 may be able to identify a small proof quickly because it performed the right case splits early. But if Z3 does not start with the right assignments or case splits it is possible that search may take a long time. The input format influences the initial assignments and case split order.
>
> *Z3: Theorem Prover, Frequently Asked Questions*[1]

In assertion examples that include complex formulae describing sets where a combination of nesting and projections are used, the SMT may just not be able to provide a definite answer. This is often the case in Yices [56]. In such cases there is no way to proceed with the verification unless there is a mechanism available to manually prove (sub-)entailments with the application of lemmas. As well as being necessary to solve this issue, lemmas also help in the cases where a sufficiently long timeout is undesirable and it may be efficient to use a divide-and-conquer approach. Quite often the SMT solver is able to prove more quickly a smaller fragment of an entailment, compared to the time it takes in the context of a larger entailment.

The extension described in this section is for lemmas that primarily serve two different purposes. The first class tackles the above problem with SMT limitations. The second allows one to show entailments between instances of recursive predicates, which cannot be performed "in-line" in a procedure's annotations because of the recursion. Note that termination of lemmas is not checked automatically and is therefore the responsibility of the human verifier.

[1] http://research.microsoft.com/en-us/um/redmond/projects/z3/old/faq.html

2.1.1 Syntax

In the same way as VeriFast [39], Lemmas here are in effect procedures, and indeed are handled in the same way by the verification tool. The only difference is that the syntax is reduced in terms of the permitted body statements (only ghost statements, skip, and if-then-else), and the guard of if-statements may contain auxiliary variables from specifications. This restricted syntax is to ensure that the lemmas have no computational effect and the pre- and post-condition is proved to be an implication. That is for a lemma $\mathcal{L}$, if $\forall \vec{a}.\, \{P\}\, \mathcal{L}(\vec{t})\, \{Q\}$ then $P \Rightarrow Q$. Their declarations are made using a new keyword, and they may also be declared as abstract with no body in which case the proof must be done outside of the tool:

$$
\begin{array}{l}
\mathsf{lemma}\ \mathcal{L}(x^*) \\
\forall [x|\alpha|@s]^*. \\
\quad \mathsf{pre} : P \\
\quad \mathsf{post} : Q \\
\quad \{\ C_{\mathcal{L}}\ \}
\end{array}
$$

The subset of the program statements that may appear in the body are as follows:

$$
\begin{array}{rrcl}
\text{ghost stmt.} & G & ::= & \ldots \mid \mathsf{lemma}\ \mathcal{L}()\ \text{inst-hints}^* \\
\text{lemma stmt.} & C_{\mathcal{L}} & ::= & \mathsf{skip} \mid C_{\mathcal{L}};\ C_{\mathcal{L}} \mid \mathsf{ghost}\ G \\
 & & & \mid \mathsf{if}\ e[= \mid \neq \mid < \mid \leq] e\ \mathsf{then}\ C_{\mathcal{L}}\ \mathsf{else}\ C_{\mathcal{L}}
\end{array}
$$

For the application of a lemma, which takes place at some point during verification of a procedure's body, a new ghost statement is introduced. The relevant symbolic execution rule is given below, which is almost identical to the rule for procedure calls except for the side condition classifying $\mathcal{L}$ as a lemma.

GhostLemma

$$
\frac{\begin{array}{c}
(\forall \vec{a}.\, \{P\} \cdot (\vec{t})\, \{Q\})[\vec{b} \backslash \vec{y}] \vdash_{\text{find-post}} \{\Upsilon\} \cdot (\vec{t'}) \left\{ \bigvee_{i=1}^{m} \exists \vec{v}_i.\Phi \right\} \\
\Pi; \forall \vec{a}, \vec{b}.\, \{P\}\, \mathcal{L}(\vec{t})\, \{Q\}\, , \Gamma \triangleright \left\{ \bigvee_{i=1}^{m} \Phi[\vec{v}_i \backslash \vec{v_i'}] \right\} C\, \{Q\}
\end{array}}{\Pi; \forall \vec{a}, \vec{b}.\, \{P\}\, \mathcal{L}(\vec{t})\, \{Q\}\, , \Gamma \triangleright \{\Upsilon\}\, \mathsf{ghost}\ \text{`}\mathsf{lemma}\ \mathcal{L}()\ \vec{b} = \vec{y}\text{'};\ C\, \{Q\}}
$$

$\vec{v}i_i$ fresh, $\mathcal{L}$ declared a lemma

2.1.2 Pure reasoning

To demonstrate the application of lemmas in the context of supplementing the power of the SMT solvers for pure reasoning, an annotated example is given in Figure 2.1. The first

lemma $proj\_subset()$
$\forall\, x, \%B, \%A.$
  pre : $\mathrm{proj}_3(\%A) \subseteq \{x\} \star \%B \subseteq \%A$;
  post : $\mathrm{proj}_3(\%B) \subseteq \{x\}$;
{ skip } // skip ensures lemma proved automatically

proc $g(x)$
$\forall\, \%S, \ldots.$
  pre : $\$\mathrm{P}(x; \%S) \star \mathrm{proj}_3(\%S) \subseteq \{x\}$;
  post : ...;
{ ... }

proc $f()$
  pre : ...;
  post : ...;
{
  ⋮ // where S_1, S_2 are large assertions:
  State 1: $\{S_1 \star \$\mathrm{P}(a; \%B) \star \mathrm{proj}_3(\%A) \subseteq \{a\} \star \%B \subseteq \%A \star S_2\}$
  ghost 'lemma $proj\_subset()$';
  State 2: $\{\mathrm{proj}_3(\%B) \subseteq \{a\} \star S_1 \star \$\mathrm{P}(a; \%B) \star \mathrm{proj}_3(\%A) \subseteq \{a\} \star \%B \subseteq \%A \star S_2\}$
  call $g(a)$;
  ⋮
}

Figure 2.1: Using lemmas to assist in pure reasoning

item is a lemma that will be used to trivialize an entailment. Because the specification represents a straight-forward pure implication, the body is simply skip. The second and third are procedures, with the latter invoking the former and it is this invocation that can be the problem. The shaded parts are from the proof tree for demonstration purposes, and are not proper annotations.

Consider the symbolic heap in State 1, and assume that S_1 and S_2 abbreviate a large number of conjunctions of atomic formulae. Amongst the large assertion, three pertinent formula are visible: a predicate instance and two subset operations. The next program statement is the call to procedure g, whose specification requires the predicate instance and that the third projection of its second argument is a subset of the singleton set of the first argument. When the x and $\%S$ in the specification are instantiated by the a and $\%B$ in State 1, the predicate can be cancelled out and the entailment to be proved for the call

is:

$$S_1 \star \text{proj}_3(\%A) \subseteq \{a\} \star \%B \subseteq \%A \star S_2 \Rightarrow \text{proj}_3(\%B) \subseteq \{a\} \tag{2.1}$$

which, to the human, is obviously true. However, if the additional state contained in S_1 and S_2 is large enough the SMT solver may not be able to generate a model before the timeout is reached. Now consider the lemma *proj_subset*, whose pre-condition includes syntactically identical formulae from State 1. By applying this lemma before the call, the lemmas post-condition will be introduced to the symbolic state (see State 2), which will produce the formula that is syntactically identical to what is required in the right-hand-side of the (2.1). The lemma can be proved automatically when the entailment is small.

Note that for simplicity the variable names used in the lemma are identical to those in the assertion to which it is applying. In reality, they may be different and instantiation hints may need to be provided, in the same way as available to call and eval commands, to ensure the desired variables are used.

The above example is fairly simple and it may be required to perform more than one step to introduce an explicit formula into the symbolic state if the lemma can't be proved. This can be handled by having the lemma's body complete the proof by applying a secondary lemma to show a possible intermediate step. The key idea in this case is to keep simplifying the problem until it is reduced to an implication that the SMT-solver *can* prove. In general the solvers are always able to handle entailments of syntactically identical formulae.

2.1.3 Predicate reasoning

The second common use for lemmas is for proving implications which use recursive predicates, and the required facts are hidden in the definitions. To demonstrate a possible scenario, see the predicate definitions and lemma in Figure 2.2.

The predicate $\%GenL$ is a standard linked list structure, whose contents is represented by the set in the second parameter. The second predicate $L also describes a linked list, however it includes an additional constraint that ensures that elements are all greater-than zero. During the verification of a program it may be necessary to show that the predicate $L implies the more general version, for instance if a library of general list operations is provided such as a list dispose. The specification of the lemma *make_generalized* reflects this implication.

In order to prove this lemma, the entire list must be unfolded and then refolded using the different predicate. The body does exactly this, first unfolding the predicate in the pre-condition. If the list is empty, then the list can be simply folded because the empty cases of each list are identical. If an element was in the list, then the lemma must be recursively applied to the rest of the list. The recursive call will produce the tail as the generalized predicate instance, and this can be folded with the current element to create the predicate in the post-condition. Due to the ability of lemma bodies to refer to auxiliary variables, no assignments or dereferences are necessary and hence there has been no computational

```
recdef $GenL(a; %L) :=
   a = 0 ⋆ %L = ∅
∨  ∃ d, n, %rest.  a ↦ d, n ⋆ $GenL(n; %rest) ⋆ %L = {d} ∪ %rest;

recdef $L(a; %L) :=
   a = 0 ⋆ %L = ∅
∨  ∃ d, n, %rest.  a ↦ d, n ⋆ $L(n; %rest) ⋆ %L = {d} ∪ %rest ⋆ 0 < d;

lemma make_generalized()
∀ ptr, %L.
   pre : $L(ptr; %L);
   post : $GenL(ptr; %L);
{
   ghost 'unfold $L(ptr; %L)';
   if %L = ∅ then {
      skip
   } else {
      ghost 'lemma make_generalized() ptr = n2'
   };
   ghost 'fold $GenL(ptr; %L)'
}
```

Figure 2.2: Using lemmas to prove properties of user-defined recursive predicates

effect. The instantiation hint with the recursive call ($ptr = n_2$) shows how the variables can be instantiated with a variable that isn't ordinarily directly accessible by procedures. The name n_2 is the unique name of the variable that is created after the skolemization of the unfolded list.

A further example of using lemmas with predicate definitions is given in Section 2.5, where a concept of non-heap-based list predicates are presented which allow more complex specifications.

2.1.4 Soundness

Before showing soundness of the new symbolic execution rule, it is first shown that a lemma declaration will prove its specification to be implication. Next, one of the judgements of the entailment prover is extended for handling lemmas so that it is assured that the symbolic state before the lemma application implies the state afterwards.

Lemma 2 (Lemma body implication)**.** *For a lemma procedure* $\mathcal{L} \in \Gamma$*,*
if $\forall \vec{a}.\, \{P\}\, \mathcal{L}(\vec{t})\, \{Q\}$ *then* $P \Rightarrow Q$*.*

Proof. By structural induction on the restricted syntax, $C_{\mathcal{L}}$ defined earlier, the implication is straightforward from the soundness of the symbolic execution rules, the pertinent parts being as follows:

- skip : Core axiom SKIP, $\{P\}\,\mathsf{skip}\,\{P\}$, can be used to derive low-level rule. To show $\{P\}\,\mathsf{skip}\,\{Q\}$, rule CONSEQUENCE requires $P \Rightarrow P$ and $P \Rightarrow Q$, the latter being the desired implication.
- $C_{\mathcal{L}1}; C_{\mathcal{L}2}$: By sequential composition (SCOMP) and transitive implication from assumptions $\{P\}\,C_{\mathcal{L}1}\,\{R\}$ and $\{R\}\,C_{\mathcal{L}2}\,\{Q\}$ which give $P \Rightarrow R$ and $R \Rightarrow Q$ respectively.
- $\mathsf{ghost}\ G$: By the interpretation of ghost statements as skip.
- $\mathsf{if}\ e_1 \bowtie e_2\ \mathsf{then}\ C_{\mathcal{L}1}\ \mathsf{else}\ C_{\mathcal{L}2}$:
  By (IF) and assumptions that $\{P \wedge e_1 \bowtie e_2\}\,C_{\mathcal{L}1}\,\{Q\}$ and
  $\{P \wedge e_1 \not\bowtie e_2\}\,C_{\mathcal{L}2}\,\{Q\}$ give $P \wedge e_1 \bowtie e_2 \Rightarrow Q$ and $P \wedge e_1 \not\bowtie e_2 \Rightarrow Q$, logical reasoning yields desired $P \Rightarrow Q$.

□

It is also necessary to include a strengthened version the $\vdash_{\text{find-post}}$ judgement. This judgement is used by the verifier to generate a post-condition Q from an invocation. The strengthened form provides an additional assurance that if there is an implication between the pre- and post-condition on the left-hand side, then that implication is preserved for the specification on the right.

Theorem 3 (Extended find-post judgement)**.** *If* $\forall\vec{a}.\,\{A\}\cdot(\vec{p})\,\{B\} \vdash_{\textit{find-post}} \{\Phi\}\cdot(\vec{t})\,\{Q\}$ *and* $A \Rightarrow B$ *then* $\models \forall\vec{a}.\,\{A\}\,k(\vec{p})\,\{B\} \Rightarrow \{\Phi\}\,k(\vec{t})\,\{Q\}$ *and* $\Phi \Rightarrow Q$
where $k \notin fv(\Phi, Q, \forall\vec{a}.\,\{A\}\cdot(\vec{p})\,\{B\})$.

Proof. The soundness proof for the original $\vdash_{\text{find-post}}$ judgement [52] shows already $\Phi \Rightarrow A$ and $B \Rightarrow Q$. With the addition of the condition in the new judgement that $A \Rightarrow B$, the additional implication in the "output" is transitively given. □

The soundness of the new (GHOSTLEMMA) rule is now shown.

Theorem 4 (Lemma application soundness)**.**

$$\textsc{GhostLemma}\quad \frac{(\forall\vec{a}.\,\{P\}\cdot(\vec{t})\,\{Q\})[\vec{b}\backslash\vec{y}] \vdash_{\textit{find-post}} \{\Upsilon\}\cdot(\vec{t'})\left\{\bigvee_{i=1}^{m}\exists\vec{v}_i.\Phi\right\} \quad \Pi;\forall\vec{a},\vec{b}.\,\{P\}\,\mathcal{L}(\vec{t})\,\{Q\}\,,\Gamma \triangleright \left\{\bigvee_{i=1}^{m}\Phi[\vec{v}_i\backslash\vec{v}'_i]\right\} C\,\{Q\}}{\Pi;\forall\vec{a},\vec{b}.\,\{P\}\,\mathcal{L}(\vec{t})\,\{Q\}\,,\Gamma \triangleright \{\Upsilon\}\,\mathsf{ghost}\ \text{`}\mathsf{lemma}\ \mathcal{L}()\ \vec{b} = \vec{y}\text{'};\ C\,\{Q\}}$$

$\vec{v}i_i$ fresh, $\mathcal{L}$ declared a lemma

Proof. Using the fact that $\mathcal{L}$ is a lemma and Lemma 2, one has $(P \Rightarrow Q)$. Using this fact with the first premise and soundness of the new $\vdash_{\text{find-post}}$ (Theorem 3), one has

$$\forall(\vec{a}.\,\{P\}\,k(\vec{t})\,\{Q\})[\vec{b}\backslash\vec{y}] \Rightarrow \{\Upsilon\}\,k(\vec{t}')\left\{\bigvee_{i=1}^{m}\exists\vec{v}_i.\Phi\right\} \quad \text{and} \quad \Upsilon \Rightarrow \bigvee_{i=1}^{m}\exists\vec{v}_i.\Phi \tag{2.2}$$

for a fresh k.

By the soundness of the symbolic execution rules and the second premise, one has

$$\Pi; \forall\vec{a},\vec{b}.\,\{P\}\,\mathcal{L}(\vec{t})\,\{Q\}\,\Gamma \models \left\{\bigvee_{i=1}^{m}\Phi[\vec{v}_i\backslash\vec{v}_i']\right\} C\,\{Q\}$$

which, by (SKOLEMIZE) and freshness of $\vec{v}_i'$ is:

$$\Pi; \forall\vec{a},\vec{b}.\,\{P\}\,\mathcal{L}(\vec{t})\,\{Q\}\,\Gamma \models \left\{\bigvee_{i=1}^{m}\exists\vec{v}_i.\Phi\right\} C\,\{Q\} \tag{2.3}$$

Because the ghost statements have no computational effect, they can be interpreted as skip. Therefore to show the conclusion it is required to prove

$$\Pi; \forall\vec{a},\vec{b}.\,\{P\}\,\mathcal{L}(\vec{t})\,\{Q\}\,,\Gamma \models \{\Upsilon\}\,\mathsf{skip};\ C\,\{Q\}$$

which, by sequential composition (SCOMP) and (2.3) it will suffice to show

$$\Pi; \forall\vec{a},\vec{b}.\,\{P\}\,\mathcal{L}(\vec{t})\,\{Q\}\,,\Gamma \models \{\Upsilon\}\,\mathsf{skip}\left\{\bigvee_{i=1}^{m}\exists\vec{v}_i.\Phi\right\}$$

This can be easily derived by the following instance of the core axiom

$$\{\Upsilon\}\,\mathsf{skip}\,\{\Upsilon\}$$

using (CONSEQUENCEPROCEDURES) and the second part of (2.2).

□

2.2 String type

The verification system presented in the collaborative work of [57] included only the integer type for program variables. Whilst this is enough for many examples, it is obviously beneficial to support other types. The supplementary type presented in this section is for strings. It will become clear in Chapters 3 and 4 that strings are particularly useful when reasoning about programs that use reflection, where loaded classes can be searched by their name.

Before detailing the built-in support for strings, an alternative approach using the original language features is considered. That is to assume some encoding from strings to integers, and leave the implementation of this encoding abstract. This approach was used in earlier work [58], where strings and a string concatenation operation were supported

forall $Concat(_, _, _).
// Append string snd onto fst. Store result in the cell at res
proc abstract $concat(fst, snd, res)$
  pre : $res \mapsto \_$;
  post : $\exists s.\ res \mapsto s \star \$\text{Concat}(fst, snd, s)$;

// Repeated appension of the same strings gives the same result
lemma abstract $concat\_equal()$
$\forall a, b, c, d.$
  pre : $\$\text{Concat}(a, b, c) \star \$\text{Concat}(a, b, d)$;
  post : $c = d$;

// Appending two distinct strings onto a common prefix yields distinct results
lemma abstract $concat\_distinct()$
$\forall x, a, b, c, d.$
  pre : $\$\text{Concat}(x, a, b) \star \$\text{Concat}(x, c, d) \star a \neq c$;
  post : $b \neq d$;

Figure 2.3: Encoding the behaviour of string concatenation in language with only integers

through the use of abstract lemmas and abstract predicates, using only the original features of the language. An excerpt of the approach can be seen in Figure 2.3, where the $Concat predicate is used to represent concatenation of its first two arguments and the result being the third argument. By using universal quantification, no definition has been provided. The specification of the abstract procedure *concat* then describes the concatenation taking place, where some result s has been generated. The two lemmas are used to describe some properties of the concatenation: 1) concatenating the same pair of input values always gives the same output value, and 2) concatenating two different suffixes to a common prefix yields different results.

Due to the abstract nature, using this approach introduces a weakness into the verification result as it is only sound if the abstract parts are proved externally. Additionally, there is an extra burden of the indirect concatenation via a procedure and the need to apply lemmas throughout the proofs. These disadvantages can be overcome by supporting the type of strings.

At this stage, the system has been extended to support: string variables, string constants, string equality, string inequality, and a concatenation operator. Like integers, the underlying reasoning is delegated to an SMT-solver. As with the set operators, it would be possible to add other operators on a by-need basis assuming they can be suitably encoded as assertions that the SMT solver can reason with.

The key steps for enriching the system with strings are to first add string expressions

to the programming and assertion languages. Next, support for strings needs to be added to the pure reasoning capabilities. Because all the necessary pure reasoning—deciding problems such as string equality—is delegated to an external solver, the addition of strings is not invasive on the original system.

In terms of the semantics in the underlying logic, the same approach as used for stored code has been taken. The heap is a function from addresses in $\mathbb{N}$ to integers $\mathbb{Z}$. To support higher-order store, code is actually stored via a simple encoding ($\lceil \_ \rceil$) to integer. Therefore an equivalent encoding from string to integer is also assumed, which maintains the same "flat" type of the heap in the semantics. Performing an eval operation on a cell containing a string is therefore handled in the same way as if the cell contained an un-encoded integer.

$$\begin{array}{rcl} \text{string expr. } e_T & ::= & @s \mid \text{“}\texttt{str}\text{”} \mid e_T{+}{+}e_T \\ \text{C} & ::= & \ldots \mid @s := e_T \mid [a] := e_T \\ \text{argument } t & ::= & \ldots \mid @x \\ \text{stmt} & ::= & \ldots \mid \mathsf{if}\ e_T\ [= \mid \neq]\ e_T\ \mathsf{then}\ C\ \mathsf{else}\ C \end{array}$$

Figure 2.4: Programming language extended with strings

The extensions to the programming language are in Fig. 2.4. This includes the sorts of variables that may appear as procedure parameters and arguments, which now include string variables. To avoid the burden of type-inference, we introduce a new set of names for strings. These are presented here as $@s$. The symbol $++$ is the concatenation operator, and strings can be stored on the heap.

$$\begin{array}{rcl} \text{element expr. } e_E & ::= & \ldots \mid e_T \\ \text{behavioural spec. } B & ::= & \forall [x|\alpha|@s]^*.\, \{P\} \cdot (t^*)\, \{Q\} \\ \text{content spec. } \mathscr{C} & ::= & \ldots \mid e_T \\ \text{atomic stmt. } A & ::= & P({e_V}^*, {e_S}^*, {e_T}^*) \mid \ldots \mid e_T = e_T \mid e_T \neq e_T \\ \text{assertion disjunction } \Psi & ::= & \exists [x|\alpha|@x]^*.\, \Theta \end{array}$$

Figure 2.5: Assertion language extended with strings

The changes to the assertion language are given in Fig. 2.5. Set elements may now also consist of string expressions, and predicates may contain string arguments. Additionally it is possible to existentially quantify over string variables in assertions and nested Hoare-triples.

The other changes relate to user-defined predicate declarations, where the parameters may include strings:

$$\mathsf{recdef}\ \mathrm{P}(x^*; \alpha^*; @s^*) = P \quad \text{and} \quad \mathsf{forall}\ [\mathsf{pure}]\ \mathrm{P}(\_^*; \_^*; \_^*)$$

and (abstract) procedure declarations where the universal quantifier may now include string variables ($\forall[x|\alpha|@x]^*$).

As mentioned above, because all the pure reasoning is delegated to the SMT-solver, and the underlying semantics remain intact except the additional cases for the interpretation of assertions, there is little else to consider. The main volume of work therefore is in the low-level implementation of the verifier, which are those straight-forward additions to the expression and variable types above, and extension of existing functions throughout the implementation to support them. The encoding of strings for the SMT-solver is included in Section 2.6, which gives a full description of the encoding for the entire new assertion language.

2.3 Sets-of-sets

Another limitation of the original Crowfoot verification system is that there were only simple one-dimensional sets. This was suitable for examples where the sets were only used in describing simple lists. However, it can be useful to provide specifications for programs which use lists-of-lists. This is the case with the metadata representation in Chapter 3, where a list of classes contains a list of methods. Therefore the natural extension has been to allow for sets-of-sets, including the appearance of sets in tuple elements. As with other pure reasoning, the onus is delegated to the SMT solver, and the extension is largely syntactic requiring the simple alteration to the class of element-expressions

$$\text{element expr. } e_E \quad ::= \quad \ldots \mid e_S$$

such that it now includes set expressions (e_S). Some example formulae showing the appearance of sets nested in elements are in Figure 2.6. Also there is an example linked list description, \$L, which itself contains a list \$L2. The set representing the top-level list is then a tuple, containing some data x, the pointer to the child list b, and the set representing that sub-list $\%B$.

Again, the encoding of assertions to the SMT-solver described in Section 2.6 includes this extension.

2.4 Anti-frame rule support

2.4.1 Alterations to the language

As discussed in Chapter 1, the "anti-frame" rule [21] for higher-order separation logic provides another mechanism for hiding. Where the (deep) frame rule allows the hiding of portions of the heap that remain untouched, the anti-frame rule allows the hiding of an invariant which is only maintained locally and so can be hidden externally from some scope. As such, outside the scope the invariant is invisible and must always hold. Inside the scope of the antiframe, the invariant is visible and may also be violated so long as it is re-established before leaving the scope.

Examples of formula containing nested sets:
$\%A = \{\%B\} \cup \{\%C\} \cup \{\%D\}$
$\%A = \%B \cup \{(a, b, \%C)\}$
$\%A = \{(a, \{x\} \cup \{y\})\}$

Example of a nested list definition:
recdef $\$\mathrm{L}(a; \%A) := a = 0 \star \%A = \emptyset$
$\vee \exists x, b, n, \%B, \%rest.\ a \mapsto x, b, n \star \$\mathrm{L2}(b; \%B) \star \%A = \{(x, b, \%B)\} \cup \%rest \star \$\mathrm{L}(n; \%rest);$

recdef $\$\mathrm{L2}(a; \%A) := a = 0 \star \%A = \emptyset$
$\vee \exists x, n, \%rest.\ a \mapsto x, n \star \%A = \{(a, x)\} \cup \%rest \star \$\mathrm{L2}(n; \%rest);$

Figure 2.6: Examples of assertions describing multi-dimensional sets

There was also a generalized version, where it was possible for the invariant "evolve", so long as the change was in some sense monotonic. When the generalized antiframe rule was first introduced in [59], it was done in two stages. The difference between the original and the two generalizations is largely to do with the strength of the invariant. In the original antiframe rule [21], the invariant could only describe the *type* of cell's content. The first generalization of the antiframe rule allows the invariant to describe the *content* of a cell. The Crowfoot system, however, is not the type-capability system as Pottier's, and this means that the first generalization can already be achieved from the implementation of the original (un-generalized) version. The original invariant might be $x \mapsto \_$. The first generalization can already be written as $x \mapsto i$ and that i can be individually universally quantified as required for each nested triple.

Before going in to details of how the anti-frame rule support has been implemented, an example is given in Figure 2.7. This example is a translated version of Pottier's example of "untracked references" [21]. The first two procedures are an accessor and mutator for a cell passed as their first argument, with the obvious specifications. These are then used by *mk_uref* to return a pair of heap-based procedures that manipulate a newly allocated cell. The body of *mk_uref* first allocates the reference cell, and then creates the code on the heap for the two procedures using partial application to fix the address of the reference cell. Looking at the specification of this procedure, the reference cell is not visible, either at the top level nor inside the nested specifications of the created getter and setter. This cell is only needed locally by the getter and setter and so can be hidden to external procedures. The way to applying the antiframe to hide this cell is to use the ghost statement annotation in the body.

A client can then use this encoding of untracked references, created by *mk_uref*, without carrying around the internal reference cell that will not be directly dereferenced. This is demonstrated in *main* which creates an untracked reference, uses the resulting proce-

```
proc uget(ptr, res)
∀ val.
  pre : ptr ↦ val ⋆ res ↦ _;
  post : ptr ↦ val ⋆ res ↦ val;
{ [res] := [ptr] }
```

```
proc uset(ptr, val)
  pre : ptr ↦ _;
  post : ptr ↦ val;
{ [ptr] := val }
```

```
proc mk_uref(v, res)
  pre : res ↦ _;
  post : ∃ fs.res ↦ fs ⋆ fs ↦ ∀ res. {res ↦ _}_(res){res ↦ _}
    ⋆ fs + 1 ↦ ∀ newVal.{emp}_(newVal){emp};
{
   locals funs, r;
   r := new v;   // This is the reference cell that we want to be untracked
   ghost 'antiframe r ↦ _';
   funs := new 0, 0;
   [funs] := uget(r, _);   // partial application used to define getter and setter for r
   [funs + 1] := uset(r, _);
   [res] := funs
}
```

```
proc main(res)
  pre : res ↦ _, _;
  post : ∃ a.res ↦ _, _;
{
   locals uref;
   call mk_uref(6, res);
   uref := [res];   // A pair of functions, no cell for the value visible
   eval[uref + 0](res);   // get: Result should be 6
   eval[uref + 1](4);   // set
   eval[uref + 0](res + 1);   // get: Result should be 4

   dispose uref + 0; dispose uref + 1   // No reference cell to dispose of
}
```

Figure 2.7: Using the antiframe rule to describe untracked references [21]

```
recdef $I(x, i) := x ↦ i;

proc m(x, f)
∀ i.
  pre : $I(x, i) ⋆ f ↦ ∀ i.{$I(x, i)}_(){$I(x, i)};
  post : $I(x, i) ⋆ f ↦ ∀ i.{$I(x, i)}_(){$I(x, i)};
{
   locals vOne, vTwo;
   ghost 'unfold $I(x, i)';
   [x] := [x] + 1;
   vOne := [x]; ghost 'fold $I(x, ?)';
   eval [f](); ghost 'unfold $I(x, ?)';
   vTwo := [x];
   ghost 'lemma assertEq(vOne, vTwo)'; // Is content at x maintained by call to f?
   tmp := [x];
   [x] := [x] − 1;
   ghost 'fold $I(x, i)'
}

proc mk(res)
  pre : res ↦ _;
  post : res ↦ ∀ f.{f ↦ {emp}_(){emp}}_(f){f ↦ {emp}_(){emp}};
{
   locals x;
   x := new 0; ghost 'fold $I(x, 0)';
   ghost 'antiframe ∃ i. $I(x, i)';
   [res] := m(x, _)
}

lemma abstract assertEq(x, y)
  pre : x = y;
  post : emp;
```

Figure 2.8: Translated version of "callee-save register" example [59]

dures, and then deallocates them. Note that only two dispose operations are needed, one for the getter and one for the setter, and it is not possible to dispose of the reference cell because it is hidden.

The example in Figure 2.8 demonstrates how Pottier's first generalization of the antiframe rule is already supported by the addition of the invariant annotation, thanks to the flexibility in the assertion language of Crowfoot. It also demonstrates how a predicate

may be used to act as a parametrized assertion variable. The example is again taken from [59] and translated in the Crowfoot language.

The procedure m uses a stored procedure at address f which maintains the invariant that states that x points to a cell containing i. During the body of m, the content of x is incremented before running the code at f and decremented after f returns. The two variables *vOne* and *vTwo* keep a record of the content of x before and after f is run, and it is checked if they are equal afterwards, i.e. that f has not violated the invariant. The interesting point is that the invariant has been temporarily broken, however the fact that the specification of f's behaviour uses its own quantification over (a different) i means that the invariant is maintained at well-balanced call-levels.

In terms of changes to the programming language, the only extra piece of syntax that was seen in these examples was a new annotation for declaring an antiframe invariant. This is the equivalent of Pottier's "hide I outside of ...", and like the other inline annotations takes the form of a ghost statement:

$$\text{ghost statement} \quad G \qquad ::= \qquad \ldots \mid \mathsf{antiframe}\ \Psi$$

2.4.2 Verification

$$\frac{\Pi; \Gamma \Vdash \{P \otimes \Psi\}\, C\, \{Q \circ \Psi\}}{\Pi; \Gamma \Vdash \{P\}\, C\, \{Q\}}\ \textsc{Antiframe}$$

Figure 2.9: Core antiframe rule

The core anti-frame rule is given in Figure 2.9. This is adapted to Crowfoot's logic from Pottier's rule for a type and capability system.

In Figure 2.10, the low-level rules implemented in the automated prover are given. This makes use of the following extended symbolic execution judgement which handles the "subtraction" of invariants.

$$\frac{\Pi; \Gamma \rhd^{-\Psi} \{\Upsilon \otimes \Psi\}\, C\, \{Q\}}{\Pi; \Gamma \rhd \{\Upsilon\}\, \mathsf{ghost\ antiframe}\ \Psi;\ C\, \{Q\}}\ \textsc{GhostAnti}$$

$$\frac{\Pi : \Upsilon \ \backslash\!\circ^{\exists \vec{w}'.\Upsilon'[\vec{v}'\backslash\vec{w}']}\ \hat{\Upsilon} \qquad \hat{\Upsilon} \vdash^{I} \exists \vec{w}_i.\ \Phi_i[\vec{v}_i\backslash\vec{w}_i] \star \Theta^{pure}}{\Pi; \Gamma \rhd^{-\exists \vec{v}'.\Upsilon'} \{\Upsilon\}\, \mathsf{skip}\, \{\exists \vec{v}_1.\Phi_1 \vee \ldots \vee \exists \vec{v}_n.\ \Phi_n\}}\ \begin{array}{l}\textsc{SkipAnti} \\ \vec{w}_1, \ldots \vec{w}_n, \vec{w}' \text{ fresh} \\ i \in \{1, \ldots, n\}\end{array}$$

Figure 2.10: Low-level rules for using the antiframe rule

$$\textsc{DiffSconjPred}\quad \frac{\big(\mathrm{P}(\vec{x},\vec{y}) \Leftrightarrow \mathrm{Q}(\vec{x}) \circ \exists \vec{v}'.\Upsilon'\big) \in \Pi \qquad \Upsilon'[\vec{v}'\backslash\vec{w}'] \vdash^{I} \exists \vec{w}.\Upsilon[\vec{v}\backslash\vec{w}] \star \mathsf{emp} \qquad \Upsilon[\vec{v}\backslash I(\vec{v})] \vdash^{\emptyset} \Upsilon'[\vec{v}'\backslash\vec{w}] \star \mathsf{emp} \qquad \Pi : \Phi \;\; \backslash\!\otimes^{\exists \vec{v}.\Upsilon} \;\hat{\Phi}}{\Pi : \mathrm{P}(v_1,\ldots,v_n) \star \Phi \;\; \backslash\!\circ^{\exists \vec{v}.\Upsilon} \; \mathrm{Q}(v_1,\ldots,v_m) \star \hat{\Phi}} \quad \vec{v} \cap \mathit{vars}(\Phi) = \emptyset,\ \vec{w}, \vec{w'} \text{ fresh}$$

$$\textsc{DiffSconj1}\quad \frac{\Phi \vdash^{I} \exists \vec{v}.\Upsilon \star \Theta \qquad \Upsilon[\vec{v}\backslash I(\vec{v})] \star \Theta \vdash^{\emptyset} \Phi \star \mathsf{emp} \qquad \Pi : \Theta \;\; \backslash\!\otimes^{\exists \vec{v}.\Upsilon} \;\hat{\Phi}}{\Phi \;\; \backslash\!\circ^{\exists \vec{v}.\Upsilon} \;\hat{\Phi}} \quad \vec{v} \cap \mathit{vars}(\Phi) = \emptyset$$

Figure 2.11: Rules of the $\circ$-subtraction judgement

Definition 5 (Symbolic execution with invariant hiding)**.** If $\Pi, \Gamma \triangleright^{-\Psi} \{P\}\, C\, \{Q\}$ then $\Pi, \Gamma \models \{P\}\, C\, \{Q \circ \Psi\}$.

The majority of existing rules for the $\triangleright$ judgement can be extended with this extra annotation and, thanks to the continuation style approach, the invariant is merely "passed through". This will require only trivial alterations to the soundness proofs. The only rule that will need to directly consider the invariant in the soundness proof is Incons, the derivation of which is straightforward because a suitable $Q \circ \Psi$ can be chosen to be used with False. Note that when there is no invariant annotation ($-\Psi$), emp can be assumed and by the definition of $\circ$ ($P \circ \mathsf{emp} \Leftrightarrow P$) the original meaning of the $\triangleright$ judgement is implied.

The first rule, GhostAnti, simply "records" the antiframe invariant by adding it as an argument to the judgement on the continuation C, and deepframes it on to any nested triples in the current symbolic heap. The second rule, SkipAnti, is a final rule for symbolic execution where the post-condition is checked, in the case where an antiframe invariant has been specified. In order to show the post-condition, the first premise generates a version of the current heap without the invariant by $\circ$-subtraction. The second premise then uses this invariant-free assertion to check entailment with the post-condition. The $\circ$-subtracted assertion is given by the $\backslash\!\circ$ judgement.

This new judgement, along with the subsidiary for $\otimes$-subtraction, is defined according to the rules in Figure 2.11 and 2.12. The intuitive meaning of the judgements are (where the shaded parts are the "generated" output):

1. If $\Phi \;\; \backslash\!\circ^{I} \;\hat{\Phi}$ then $\Phi \Leftrightarrow \hat{\Phi} \circ I$
2. If $\Phi \;\; \backslash\!\otimes^{I} \;\hat{\Phi}$ then $\Phi \Leftrightarrow \hat{\Phi} \otimes I$

with the precise definitions given later in Section 2.4.3.2.

The $\backslash\!\circ$ rules behave as follows:

- (DiffSconjPred) – If there is a predicate instance whose definition has been defined using $\circ$ with an equivalent invariant, then replace the predicate with the invariant-free predicate from the definition. As the $\star$-part of the invariant has now been removed, the rest of the assertion is handled by $\backslash\!\otimes$, rather than $\backslash\!\circ$.

$$\textsc{DiffSconj2}\quad \frac{\Pi : A \;\backslash\otimes^{\Psi}\; \hat{A} \qquad \Pi : \Phi \;\backslash\otimes^{\Psi}\; \hat{\Phi}}{\Pi : A \star \Phi \;\backslash\otimes^{\Psi}\; \hat{A} \star \hat{\Phi}}$$

$$\textsc{DiffPointsTo}\quad \frac{\begin{array}{c}\Pi : e_A \mapsto \mathscr{C}_1 \;\backslash\otimes^{\Psi}\; e_A \mapsto \hat{\mathscr{C}}_1 \\ \vdots \\ \Pi : e_A + n - 1 \mapsto \mathscr{C}_n \;\backslash\otimes^{\Psi}\; e_A + n - 1 \mapsto \hat{\mathscr{C}}_n\end{array}}{\Pi : e_A \mapsto \mathscr{C}_1, \ldots, \mathscr{C}_n \;\backslash\otimes^{\Psi}\; e_A \mapsto \hat{\mathscr{C}}_1, \ldots, \hat{\mathscr{C}}_n}$$

$$\textsc{DiffPointsToVal}\quad \frac{}{e_A \mapsto v \;\backslash\otimes^{\Psi}\; e_A \mapsto v} \quad v \in \{\_, e_V, e_T\}$$

$$\textsc{DiffPointsToTriple}\quad \frac{\Pi : P \;\backslash\circ^{\Psi}\; \hat{P} \qquad \Pi : Q \;\backslash\circ^{\Psi}\; \hat{Q}}{\Pi : e_A \mapsto \{P\} \cdot (\vec{p})\, \{Q\} \;\backslash\otimes^{\Psi}\; e_A \mapsto \{\hat{P}\} \cdot (\vec{p}) \{\hat{Q}\}}$$

$$\textsc{DiffPure}\quad \frac{}{v_1 \oplus v_2 \;\backslash\otimes^{\Psi}\; v_1 \oplus v_2} \oplus \in \{=, \neq, \in, \notin, \subseteq\}$$

$$\textsc{DiffPred}\quad \frac{\Pi : (\mathrm{P}(\vec{v}) \Leftrightarrow Q) \in \Pi}{\Pi : \mathrm{P}(e\vec{_V}, e\vec{_S}, e\vec{_T}) \;\backslash\otimes^{\Psi}\; \mathrm{P}(e\vec{_V}, e\vec{_S}, e\vec{_T})} \quad \text{where } Q \text{ is a left-zero of } \otimes$$

Figure 2.12: Rules of the $\otimes$-subtraction judgement

- (DiffSconj) – Look for the invariant in the assertion, and once found continue by $\otimes$-subtracting from the found frame/rest of the heap.

The $\backslash\otimes$ rules are mostly straightforward. Predicate handling is limited only to those whose definition does not contain any nested triples.

Rule (DiffPointsToTriple) shows how the two judgements are mutually recursive. Rule (DiffPred) may only apply to predicates who's definition does not includes any nested triples. This is due to the tools limited handling of predicate definitions, which cannot be declared with the $\otimes$ operator.

One may think that there should be an analogous version of predicate definitions for antiframing, as is already present for deepframing by way of the $\circ$ operator. For instance recdef $\$\mathrm{P}(\ldots) := \$\mathrm{R}(\ldots) \;\backslash\circ\; \Psi$, where the intention is that the definition of \$P will be \$R with invariant Ψ subtracted. While this would be easy to implement as the implementation can already handle the necessary invariant subtraction, it would be of questionable benefit. The purpose of the antiframe rule is to hide the locally maintained parts of the heap from external procedures. As such the invariant will only be appearing in one of a small local set of related procedure's specifications, and not in the specifications for rest of the program. It is thought that the majority of procedures will not "see" the invariant, and so the most commonly appearing specifications will be the smaller versions. Additionally, it is less work to write definitions where the invariant is to be automatically added.

2.4.3 Soundness

The first step to proving the soundness of the antiframe extension to the verification system is to show that the core rule is sound. Following this, the invariant-subtraction judgements are shown sound, and then the low-level rules which comprise the semi-automated prover's algorithm.

2.4.3.1 Core antiframe rule

The soundness of the antiframe rule is proved in [24], with respect to the type capability language used by Pottier in the introduction of the antiframe rule. However, the programming language used here is different and so the soundness of the core rule must be adapted.

The original semantics for interpreting Hoare-triples in the Crowfoot logic (Definition 1, Chapter 1) is changed to the following definition, where the invariant is extended by the quantified w', which will be used to absorb the antiframe invariant.

Definition 6. Let $p, q \in \mathit{Pred}$, $w \in W$, $\eta \in \mathit{Env}$, $\sigma \in \mathit{SetEnv}$, let C be a program statement and let γ be a procedure environment. We define that $w, \eta, \sigma \models^{\gamma}_{n} (p, C, q)$ holds iff the following holds: for all $r \in \mathit{UPred}(H)$, all $m < n$, all heaps h, all stacks s, if $(m, \eta, \sigma, h) \in p(w) \star i^{-1}(w)(\mathrm{emp}) \star r$, then:

1. $(C, s \cdot \eta, h) \in \mathrm{Safe}^{\gamma}_{m}$.
2. For all $k \leq m$ and all $h' \in \mathit{Heap}$, $\eta' \in \mathit{Env}$, if $(C, s \cdot \eta, h) \rightsquigarrow^{\gamma}_{k} (\mathsf{skip}, s \cdot \eta', h')$, then $(m - k, \eta', \sigma, h') \in \bigcup_{w'} q(w \circ w') \star i^{-1}(w \circ w')(\mathrm{emp}) \star r$.

The following lemma will help to abbreviate the steps later in the proof.

Lemma 7. $[\![P]\!]_{\pi}(w) \star i^{-1}(w)(\mathrm{emp}) \quad = \quad i^{-1}(i([\![P]\!]_{\pi}) \circ w)(\mathrm{emp})$

Proof. By definition of $\otimes, \circ, \star$. □

The following lemma from [24] is important for the proof, and describes "commutative pairs".

Lemma 8. *For all worlds $w_0, w_1 \in W$, there exists $w_0', w_1' \in W$ such that*

$$w_0' = i(i^{-1}(w_0) \otimes w_1'), \quad w_1' = i(i^{-1}(w_1) \otimes w_0'), \quad \textit{and} \quad w_0 \circ w_1' = w_1 \circ w_0'$$

Theorem 9 (Soundness of standard anti-frame)**.** *Suppose* $\models^{\gamma}_{\pi} (P \otimes \Psi, C, Q \otimes \Psi \star \Psi)$. *Then* $\models^{\gamma}_{\pi} (P, C, Q)$.

Proof. Using Lemma 8 for $w, i([\![\Psi]\!]_{\pi}) \in W$ there exists $w', w'' \in W$,

$$w' = i(i^{-1}(w) \otimes w''), \quad w'' = i([\![\Psi]\!]_{\pi} \otimes w') \quad \text{and} \quad i([\![\Psi]\!]_{\pi}) \circ w' = w \circ w'' \tag{2.4}$$

First, the following is shown:

$$[\![P]\!]_{\pi}(w) \star i^{-1}(w)(\mathrm{emp}) \star r \quad \subseteq \quad [\![P \otimes \Psi]\!]_{\pi}(w') \star i^{-1}(w')(\mathrm{emp}) \star r \tag{2.5}$$

which, by $\star$-monotonicity, can be broken down into two parts.

$$
\begin{array}{ll}
 & [\![P]\!]_\pi\,(w) \\
\subseteq & \{\text{ By monotonicity of the worlds }\} \\
 & [\![P]\!]_\pi\,(w \circ w'') \\
= & \{\text{ By (2.4)}\} \\
 & [\![P]\!]_\pi\,(i([\![\Psi]\!]_\pi) \circ w') \\
= & \{\text{ By definition of } \otimes\} \\
 & ([\![P]\!]_\pi \otimes i([\![\Psi]\!]_\pi))(w') \\
= & \{\text{ By definition of } \otimes\} \\
 & [\![P \otimes \Psi]\!]_\pi\,(w')
\end{array}
$$

$$
\begin{array}{ll}
 & i^{-1}(w)(\text{emp}) \\
\subseteq & \{\text{ by monotonicity of the worlds}\} \\
 & i^{-1}(w)(\text{emp} \circ w'') \\
= & \{\text{ By fact that emp is unit}\} \\
 & i^{-1}(w)(w'' \circ \text{emp}) \\
= & \{\text{ by definition of } \otimes\} \\
 & (i^{-1}(w) \otimes w'')(\text{emp}) \\
= & \{\text{ By (2.4)}\} \\
 & i^{-1}(w')(\text{emp})
\end{array}
$$

This gives (2.5). By the premise and Definition 6 it can be assumed:

For all $r \in UPred(H)$, all $m < n$, all heaps h, stacks s, if $(m, \eta, \sigma, h) \in [\![P \otimes \Psi]\!]_\pi\,(w') \star i^{-1}(w)(\text{emp}) \star r$, then

(a) $(C, s \cdot \eta, h) \in \text{Safe}^\gamma_m$

(b) For all $k \leq m$ and all $h' \in Heap$, $\eta' \in Env$, if $(C, s \cdot \eta, h) \rightsquigarrow^\gamma_k (\mathsf{skip}, s \cdot \eta', h')$, then $(m - k, \eta', \sigma, h') \in \bigcup_{\hat{w}} [\![Q \otimes \Psi \star \Psi]\!]_\pi\,(w' \circ \hat{w}) \star i^{-1}(w' \circ \hat{w})(\text{emp}) \star r$

It is required to prove: For all $r \in UPred(H)$, all $m < n$, all heaps h, stacks s, if $(m, \eta, \sigma, h) \in [\![P]\!]_\pi\,(w) \star i^{-1}(w)(\text{emp}) \star r$, then

(i) $(C, s \cdot \eta, h) \in Safe^\gamma_m$

(ii) For all $k \leq m$ and all $h' \in Heap$, $\eta' \in Env$, if $(C, s \cdot \eta, h) \rightsquigarrow^\gamma_k (\mathsf{skip}, s \cdot \eta', h')$, then $(m - k, \eta', \sigma, h') \in \bigcup_{\hat{w}'} [\![Q]\!]_\pi\,(w \circ \hat{w}') \star i^{-1}(w \circ \hat{w}')(\text{emp}) \star r$

By (a) and (2.5), the (i) is given. For (ii) it is shown that, with $\star$-monotonicity for the "frame" r, it can be derived from the implication of (b).

$$
\begin{aligned}
& [\![Q \otimes \Psi \star \Psi]\!]_\pi \, (w' \circ \hat{w}) \star i^{-1}(w' \circ \hat{w})(\text{emp}) \\
= \quad & \{ \text{ By definition of } \star \ \} \\
& [\![Q \otimes \Psi]\!]_\pi \, (w' \circ \hat{w}) \star [\![\Psi]\!] \, (w' \circ \hat{w}) \star i^{-1}(w' \circ \hat{w})(\text{emp}) \\
= \quad & \{ \text{ By definition of } \otimes \ \} \\
& [\![Q]\!]_\pi \, (i([\![\Psi]\!]_\pi) \circ w' \circ \hat{w}) \star [\![\Psi]\!]_\pi \, (w' \circ \hat{w}) \star i^{-1}(w' \circ \hat{w})(\text{emp}) \\
= \quad & \{ \text{ By Lemma 7 } \} \\
& [\![Q]\!]_\pi \, (i([\![\Psi]\!]_\pi) \circ w' \circ \hat{w}) \star i^{-1}(i([\![\Psi]\!]_\pi) \circ w' \circ \hat{w})(\text{emp}) \\
= \quad & \{ \text{ By (2.4) } \} \\
& [\![Q]\!]_\pi \, (w \circ w'' \circ \hat{w}) \star i^{-1}(w \circ w'' \circ \hat{w})(\text{emp}) \\
\Rightarrow \quad & \{ \text{ By existential introduction, setting } \hat{w}' = w'' \circ \hat{w} \ \} \\
& \bigcup_{\hat{w}'} [\![Q]\!]_\pi \, (w \circ \hat{w}) \star i^{-1}(w \circ \hat{w})(\text{emp})
\end{aligned}
$$

□

2.4.3.2 Invariant subtraction

Theorem 10. *The (mutually recursive) judgements for deeply "subtracting" invariants are sound. That is to say:*

1. *If* $\Pi : \Phi \ \setminus\!\!\circ^{\exists \vec{v}.\Upsilon} \ \hat{\Phi}$ *(where* $\vec{v} \cap fv(\Phi) = \emptyset$*) then* $\Pi \models \Phi \Leftrightarrow (\hat{\Phi} \otimes \exists \vec{v}.\Upsilon) \star \exists \vec{v}.\Upsilon$ *where* $fv(\hat{\Phi}) \subseteq fv(\Phi)$.
2. *If* $\Pi : \Phi \ \setminus\!\!\otimes^{\exists \vec{v}.\Upsilon} \ \hat{\Phi}$ *(where* $\vec{v} \cap fv(\Phi) = \emptyset$*) then* $\Pi \models \Phi \Leftrightarrow \hat{\Phi} \otimes \exists \vec{v}.\Upsilon$ *where* $fv(\hat{\Phi}) \subseteq fv(\Phi)$.

Proof. The judgements are defined by the rules in Figures 2.11 and 2.12. The rules for $\setminus\!\!\circ$ are shown to be sound below. The rules for $\setminus\!\!\otimes$ rely mostly on the distribution rules for $\otimes$, except for DiffPointsToTriple, which uses the equivalence from the $\setminus\!\!\circ$ judgement.

(DiffSconjPred)

$$\textsc{DiffSconjPred}\quad \frac{\begin{array}{c}\left(\mathrm{P}(\vec{x},\vec{y}) \Leftrightarrow \mathrm{Q}(\vec{x}) \circ \exists \vec{v}'.\Upsilon'\right) \in \Pi \\ \Upsilon'[\vec{v}'\backslash\vec{w}'] \vdash^{I} \exists \vec{w}.\Upsilon[\vec{v}\backslash\vec{w}] \star \mathsf{emp} \qquad \Upsilon[\vec{v}\backslash I(\vec{v})] \vdash^{\emptyset} \Upsilon'[\vec{v}'\backslash\vec{w}] \star \mathsf{emp} \\ \Pi : \Phi \ \setminus\!\!\otimes^{\exists \vec{v}.\Upsilon} \ \hat{\Phi}\end{array}}{\Pi : \mathrm{P}(v_1,\ldots,v_n) \star \Phi \ \setminus\!\!\circ^{\exists \vec{v}.\Upsilon} \ \mathrm{Q}(v_1,\ldots,v_m) \star \hat{\Phi}} \quad \begin{array}{l}\vec{v} \cap vars(\Phi) = \emptyset, \\ \vec{w}, \vec{w'} \text{ fresh}\end{array}$$

The soundness of $\vdash$ states:

If $\Phi \vdash^{I} \exists \vec{v}.\Upsilon \star \Theta$ (where $fv(\Phi) \cap \vec{v} = \emptyset$) then $\models \Phi \Rightarrow \Upsilon[\vec{v}\backslash I(\vec{v})] \star \Theta$ where: $fv(\Theta) \subseteq fv(\Phi)$, $dom(I) = \vec{v}$ and $fv(Im(I)) \subseteq fv(\Phi)$.

To show the two ($\vdash$) premises are well-formed, it must be shown that the existential variables on the right-hand side do not overlap with the free variables on the left. In the first instance this is the ensured by the freshness of $\vec{w}$. In the second, there are no existential variables to consider.

By the first entailment in the premise and the soundness of ($\vdash^{I}$):

$$\Upsilon'[\vec{v}'\backslash\vec{w}'] \Rightarrow \Upsilon[\vec{v}\backslash\vec{w}][\vec{w}\backslash I(\vec{w})] \star \mathsf{emp}$$

which, due to the freshness of $\vec{w}$ and applying the substitution is:

$$\Upsilon'[\vec{v}'\backslash\vec{w}'] \Rightarrow \Upsilon[\vec{v}\backslash I(\vec{w})] \star \mathsf{emp}$$

and by (emp-Unit):

$$\Upsilon'[\vec{v}'\backslash\vec{w}'] \Rightarrow \Upsilon[\vec{v}\backslash I(\vec{w})] \tag{2.6}$$

By the second entailment in the premise and the soundness of ($\vdash^{I}$):

$$\Upsilon[\vec{v}\backslash I(\vec{w})] \Rightarrow \Upsilon'[\vec{v}'\backslash\vec{w}'] \star \mathsf{emp}$$

which, by (emp-Unit) is:

$$\Upsilon[\vec{v}\backslash I(\vec{w})] \Rightarrow \Upsilon'[\vec{v}'\backslash\vec{w}'] \tag{2.7}$$

With biconditional introduction using (2.6) and (2.7), the equivalence is given:

$$\Upsilon[\vec{v}\backslash I(\vec{w})] \Leftrightarrow \Upsilon'[\vec{v}'\backslash\vec{w}']$$

which, by existential introduction implies

$$\exists\vec{v}.\Upsilon \Leftrightarrow \exists\vec{v}'.\Upsilon' \tag{2.8}$$

The last premise, and the soundness of $\backslash\!\!\otimes$ gives:

$$\Phi \Leftrightarrow \hat{\Phi} \otimes \exists\vec{v}.\Upsilon \tag{2.9}$$

It is required to prove:

$$\mathrm{P}(v_1,\ldots,v_n) \star \Phi \Leftrightarrow ((\mathrm{Q}(v_1,\ldots,v_m) \star \hat{\Phi}) \otimes \exists\vec{v}.\Upsilon) \star \exists\vec{v}.\Upsilon$$

which can be shown as follows:

$$\begin{array}{ll}
 & \mathrm{P}(v_1,\ldots,v_n) \star \Phi \\
\Leftrightarrow & \{\text{ By the first premise and } \star\text{-monotonicity }\} \\
 & (\mathrm{Q}(v_1,\ldots,v_m) \circ \exists\vec{v}'.\Upsilon') \star \Phi \\
\Leftrightarrow & \{\text{ By (2.9) and } \star\text{-monotonicity }\} \\
 & (\mathrm{Q}(v_1,\ldots,v_m) \circ \exists\vec{v}'.\Upsilon') \star \hat{\Phi} \otimes \exists\vec{v}.\Upsilon \\
\Leftrightarrow & \{\text{ By } \circ \text{ definition }\} \\
 & (\mathrm{Q}(v_1,\ldots,v_m) \otimes \exists\vec{v}'.\Upsilon') \star \exists\vec{v}'.\Upsilon' \star \hat{\Phi} \otimes \exists\vec{v}.\Upsilon \\
\Leftrightarrow & \{\text{ By } \star \text{ commutativity }\} \\
 & (\mathrm{Q}(v_1,\ldots,v_m) \otimes \exists\vec{v}'.\Upsilon') \star \hat{\Phi} \otimes \exists\vec{v}.\Upsilon \star \exists\vec{v}'.\Upsilon' \\
\Leftrightarrow & \{\text{ By (2.8) }\} \\
 & (\mathrm{Q}(v_1,\ldots,v_m) \otimes \exists\vec{v}.\Upsilon) \star \hat{\Phi} \otimes \exists\vec{v}.\Upsilon \star \exists\vec{v}.\Upsilon \\
\Leftrightarrow & \{\text{ By } \otimes \text{ distribution }\} \\
 & ((\mathrm{Q}(v_1,\ldots,v_m) \star \hat{\Phi}) \otimes \exists\vec{v}.\Upsilon) \star \exists\vec{v}.\Upsilon
\end{array}$$

(DiffSconj1)

$$\frac{\Phi \vdash^{I} \exists\vec{v}.\Upsilon \star \Theta \qquad \Upsilon[\vec{v}\backslash I(\vec{v})] \star \Theta \vdash^{\emptyset} \Phi \star \mathsf{emp} \qquad \Pi : \Theta \;\backslash\!\!\otimes^{\exists\vec{v}.\Upsilon}\; \hat{\Phi}}{\Phi \;\backslash\!\circ^{\exists\vec{v}.\Upsilon}\; \hat{\Phi}}\; \text{DiffSconj1}\; \vec{v} \cap \mathit{vars}(\Phi) = \emptyset$$

To show the first two premises ($\vdash$) are well-formed, it must be shown that the existential variables on the right-hand side do not overlap with the free variables on the left. In the first instance, this is the side-condition. In the second, there are no existential variables to consider.

By the first premise and the soundness of $(\vdash^I)$:

$$\Phi \Rightarrow \Upsilon[\vec{v}\backslash I(\vec{v})] \star \Theta \tag{2.10}$$

By the second premise and the soundness of $(\vdash^I)$:

$$\Upsilon[\vec{v}\backslash I(\vec{v})] \star \Theta \Rightarrow \Phi \star \mathsf{emp}$$

which, by (emp-UNIT) is

$$\Upsilon[\vec{v}\backslash I(\vec{v})] \star \Theta \Rightarrow \Phi \tag{2.11}$$

Together, with biconditional introduction, (2.10) and (2.11) give the equivalence:

$$\Phi \Leftrightarrow \Upsilon[\vec{v}\backslash I(\vec{v})] \star \Theta$$

and by the constraints on I and $fv(\Phi)$ from the $(\vdash^I)$ in the first premise, it is equivalent to

$$\Phi \Leftrightarrow \exists \vec{v}.\Upsilon \star \Theta \tag{2.12}$$

The third premise, and the soundness of $\not\otimes$ gives:

$$\Theta \Leftrightarrow \hat{\Phi} \otimes \exists \vec{v}.\Upsilon \tag{2.13}$$

It is required to prove $\Phi \Leftrightarrow (\hat{\Phi} \otimes \exists \vec{v}.\Upsilon) \star \exists \vec{v}.\Upsilon$, which is reasoned as follows:

$$\begin{array}{ll} & \Phi \\ \Leftrightarrow & \{ \text{ By (2.12)} \} \\ & \exists \vec{v}.\ \Upsilon \star \Theta \\ \Leftrightarrow & \{ \ fv(\Theta) \subseteq fv(\Phi) \text{ and } \vec{v} \cap fv(\Phi) = \emptyset \} \\ & (\exists \vec{v}.\ \Upsilon) \star \Theta \\ \Leftrightarrow & \{ \text{ By (2.13)} \} \\ & (\exists \vec{v}.\ \Upsilon) \star (\hat{\Phi} \otimes \exists \vec{v}.\Upsilon) \\ \Leftrightarrow & \{ \text{ Commutativity of } \star \} \\ & (\hat{\Phi} \otimes \exists \vec{v}.\Upsilon) \star \exists \vec{v}.\Upsilon \end{array}$$

□

2.4.3.3 Low-level symbolic execution rules

This section includes the soundness proofs for the two new symbolic execution rules, as implemented in the automated prover. The first is the new rule for skip, which handles the antiframe invariant when checking the post-condition. The second is for handling the invariant annotation, in the form of a ghost statement.

New skip rule

$$\frac{\Pi : \Upsilon \ \backslash\!\!\circ^{\exists \vec{w}'.\Upsilon'[\vec{v}'\backslash\vec{w}']} \ \hat{\Upsilon} \qquad \hat{\Upsilon} \vdash^{I} \exists \vec{w}_i.\ \Phi_i[\vec{v}_i\backslash\vec{w}_i] \star \Theta^{pure}}{\Pi; \Gamma \rhd^{-\exists \vec{v}'.\Upsilon'} \{\Upsilon\}\, \mathsf{skip}\, \{\exists \vec{v}_1.\Phi_1 \vee \ldots \vee \exists \vec{v}_n.\ \Phi_n\}} \ \textsc{SkipAnti}$$

$$\vec{w}_1, \ldots \vec{w}_n, \vec{w}' \text{ fresh}, \ i \in \{1, \ldots, n\}$$

The first premise and (soundness of $\backslash\!\!\circ$) gives:

$$\Upsilon \Leftrightarrow (\hat{\Upsilon} \otimes \exists \vec{w}'.\Upsilon'[\vec{v}'\backslash\vec{w}']) \star \exists \vec{w}'.\Upsilon'[\vec{v}'\backslash\vec{w}'] \tag{2.14}$$

if $fv(\Upsilon) \cap \vec{w}' = \emptyset$, which is ensured from the $\vec{w'}$ freshness side-condition.
The second premise and soundness of $(\vdash^{I})$ with the freshness of $\vec{w}_i$ gives:

$$\hat{\Upsilon} \Rightarrow \exists \vec{w}_i.\ \Phi_i[\vec{v}_i\backslash\vec{w}_i] \star \Theta^{pure} \tag{2.15}$$

To show the conclusion, it is required to prove

$$\Pi, \Gamma \models \{\Upsilon\}\, \mathsf{skip}\, \left\{(\exists \vec{v}_1.\Phi_1 \vee \ldots \vee \exists \vec{v}_n.\Phi_n) \circ \exists \vec{v}'.\Upsilon'\right\} \tag{2.16}$$

This is derived from the following instance of the core axiom

$$\{\Upsilon\}\, \mathsf{skip}\, \{\Upsilon\}$$

which, by (CONSEQUENCE) and the fact that $\Upsilon \Rightarrow \Upsilon$ and $\Upsilon \Rightarrow (\exists \vec{v}_1.\Phi_1 \vee \ldots \vee \exists \vec{v}_n.\Phi_n) \circ \exists \vec{v}'.\Upsilon'$ (shown later), becomes the required (2.16):

$$\{\Upsilon\}\, \mathsf{skip}\, \left\{(\exists \vec{v}_1.\Phi_1 \vee \ldots \vee \exists \vec{v}_n.\Phi_n) \circ \exists \vec{v}'.\Upsilon'\right\}$$

The required implication for the post-conditions is reasoned as follows:

$$
\begin{array}{ll}
 & \Upsilon \\
\Leftrightarrow & \{ \text{ By (2.14) } \} \\
 & (\hat{\Upsilon} \otimes \exists \vec{w}'.\Upsilon'[\vec{v}'\backslash\vec{w}']) \star \exists \vec{w}'.\Upsilon'[\vec{v}'\backslash\vec{w}'] \\
\Rightarrow & \{ \text{ By (2.15), and } (\star\text{-Monotonicity}), (\otimes\text{-Mono}) \} \\
 & ((\exists \vec{w}_i.\ \Phi_i[\vec{v}_i\backslash\vec{w}_i] \star \Theta^{pure}) \otimes \exists \vec{w}'.\Upsilon'[\vec{v}'\backslash\vec{w}']) \star \exists \vec{w}'.\Upsilon'[\vec{v}'\backslash\vec{w}'] \\
\Rightarrow & \{ \text{ By } \star\text{-SplitPureLeft and } (\star\text{-Monotonicity}), (\otimes\text{-Mono}) \} \\
 & ((\exists \vec{w}_i.\ \Phi_i[\vec{v}_i\backslash\vec{w}_i]) \otimes \exists \vec{w}'.\Upsilon'[\vec{v}'\backslash\vec{w}']) \star \exists \vec{w}'.\Upsilon'[\vec{v}'\backslash\vec{w}'] \\
\Leftrightarrow & \{ \text{ Rename bound } \vec{w}_i \text{ by } \vec{v}_i \text{ with Lemma 11 and } \vec{w}_i \text{ freshness } \} \\
 & ((\exists \vec{v}_i.\ \Phi_i) \otimes \exists \vec{w}'.\Upsilon'[\vec{v}'\backslash\vec{w}']) \star \exists \vec{w}'.\Upsilon'[\vec{v}'\backslash\vec{w}'] \\
\Leftrightarrow & \{ \text{ Rename both bound } \vec{w}' \text{ by } \vec{v}' \text{ with Lemma 11 and } \vec{w}' \text{ freshness } \} \\
 & ((\exists \vec{v}_i.\ \Phi_i) \otimes \exists \vec{v}'.\Upsilon') \star \exists \vec{v}'.\Upsilon' \\
\Leftrightarrow & \{ \text{ By } (\circ\text{-Definition}) \} \\
 & (\exists \vec{v}_i.\ \Phi_i) \circ \exists \vec{v}'.\Upsilon' \\
\Rightarrow & \{ \text{ By logical addition } (P \Rightarrow (P \vee Q)) \text{ and } i \in 1, \ldots, n \ \} \\
 & ((\exists \vec{v}_1.\ \Phi_1) \circ \exists \vec{v}'.\Upsilon') \vee \ldots \vee ((\exists \vec{v}_n.\ \Phi_n) \circ \exists \vec{v}'.\Upsilon') \\
\Leftrightarrow & \{ \text{ By Lemma 12 } \} \\
 & (\exists \vec{v}_1.\Phi_1 \vee \ldots \vee \exists \vec{v}_n.\Phi_n) \circ \exists \vec{v}'.\Upsilon'
\end{array}
$$

Lemma 11. *$\exists \vec{w}.P[\vec{v}\backslash\vec{w}] \Leftrightarrow \exists \vec{v}.P$ where $\vec{w} \notin fv(P)$.*

Proof. By repeated α-conversion for each $v \in \vec{v}$ and $w \in \vec{w}$. □

Lemma 12. *($\circ$ disjunction distribution) $(P \circ R) \vee (Q \circ R) \Leftrightarrow (P \vee Q) \circ R$.*

Proof. This is easily derived from ($\otimes$-Connectives) and the definition of $\circ$:

$$
\begin{array}{l}
(P \circ R) \vee (Q \circ R) \\
\Leftrightarrow \{ \ (\circ\text{-Definition}) \ \} \\
((P \otimes R) \star R) \vee ((Q \otimes R) \star R) \\
\Leftrightarrow \{ \text{ Separation logic } \star \text{ distribution law } \} \\
((P \otimes R) \vee (Q \otimes R)) \star R \\
\Leftrightarrow \{ \ (\otimes\text{-Connectives}) \ \} \\
((P \vee Q) \otimes R) \star R \\
\Leftrightarrow \{ \ (\circ\text{-Definition}) \ \} \\
(P \vee Q) \circ R
\end{array}
$$

□

Antiframe annotation

$$\frac{\Pi; \Gamma \rhd^{-\Psi} \{\Upsilon \otimes \Psi\}\, C\, \{Q\}}{\Pi; \Gamma \rhd \{\Upsilon\}\, \mathsf{ghost\ antiframe}\ \Psi;\ C\, \{Q\}}\ \textsc{GhostAnti}$$

It can be assumed from the premise, along with the soundness of $(\rhd^{-\Psi})$, that:

$$\Pi; \Gamma \models \{\Upsilon \otimes \Psi\}\, C\, \{Q \circ \Psi\}$$

By applying (ANTIFRAME) to this assumption, we have:

$$\Pi; \Gamma \models \{\Upsilon\}\, C\, \{Q\} \tag{2.17}$$

Because the ghost statement has no computational effect, it is required to prove:

$$\Pi; \Gamma \models \{\Upsilon\}\, \mathsf{skip};\ C\, \{Q\}$$

which, by (SCOMP) and (SKIP), follows from:

$$\Pi; \Gamma \models \{\Upsilon\}\, C\, \{Q\} \tag{2.18}$$

which is exactly (2.17).

2.5 Pure inductive predicates

2.5.1 Usage

Whereas the tool does have built-in support for heap-based linked list segments, sometimes it is useful for a specification to include a pure list structure to describe further functional behaviour. This could be because a common data structure might need different constraints for different usages, which is better asserted through external formula rather than contaminating the data structure's definition. A simple example is given in Figure 2.13.

Given the list structure described by the first predicate, there are two further predicates that assert different additional constraints. The first states that all the elements are zero or greater, and the second ensures elements are one of the integers $1-3$. To demonstrate how the two lists can be maintained concurrently, an example using the second type restriction is given in the procedure *cycle_all*. The function of this procedure is to recursively increment each element of a list starting at address l, cycling back to 1 if the incremented integer goes out of range. The specification includes the linked list and the specialized pure predicate, with the post-condition showing that there may be a new list after execution. The restriction that the first projections of the before and after sets which describe the list ensures that only the data is updated, and the list pointers are unchanged. The interesting part lies in the ghost annotations. First, the linked list predicate is unfolded, which may branch for its two cases. In the non-empty case, the body first makes the recursive call, before performing the desired action on the current element. In order for verification to

recdef $\$\mathrm{L}(a; \%L) := a = 0 \star \%L = \emptyset$
$\quad \vee \exists\, d, n, \%rest.\ a \mapsto d, n \star \$\mathrm{L}(n; \%rest) \star \%L = \{(a, d)\} \cup \%rest \star a \notin \mathrm{proj}_1(\%rest);$

recdef $\$\mathrm{NonNeg}(; \%L) := \%L = \emptyset$
$\quad \vee \exists\, a, i, \%rest.$
$\quad\quad \%L = \{(a, i)\} \cup \%rest \star a \notin \mathrm{proj}_1(\%rest) \star \$\mathrm{NonNeg}(; \%rest) \star 0 \leq i;$

recdef $\$\mathrm{OptThree}(; \%L) := \%L = \emptyset$
$\quad \vee \exists\, a, i, \%rest.$
$\quad\quad \%L = \{(a, i)\} \cup \%rest \star a \notin \mathrm{proj}_1(\%rest) \star \$\mathrm{OptThree}(; \%rest) \star i \in \{1\} \cup \{2\} \cup \{3\};$

proc $cycle\_all(l)$
$\forall\, \%L.$
pre : $\$\mathrm{L}(l; \%L) \star \$\mathrm{OptThree}(; \%L);$
post : $\exists\, \%newL.\ \$\mathrm{L}(l; \%newL) \star \$\mathrm{OptThree}(; \%newL) \star \mathrm{proj}_1(\%newL) = \mathrm{proj}_1(\%L);$
{
locals $cur, next$;
ghost 'unfold $\$\mathrm{L}(?; ?)$';
if $l = 0$ then {
skip
} else {
ghost 'lemma $unfold\_optThree();$'
$next := [l + 1];$
call $cycle\_all(next)$;
$cur := [l];$
$[l] := cur + 1;$
if $cur = 3$ then { $[l] := 1;$ ghost 'fold $\$\mathrm{OptThree}(; \%newL1 \cup \{(l, 1)\})$' }
else { skip; ghost 'fold $\$\mathrm{OptThree}(; \%newL1 \cup \{(l, cur + 1)\})$' }
};
ghost 'fold $\$\mathrm{L}(l; ?)$'
}

Figure 2.13: Two pure predicates describing different properties of a common base list structure

```
lemma unfold_optThree()
  ∀ a, i, %L.
 pre : $OptThree(; %L) ⋆ (a, i) ∈ %L;
 post : ∃ %rest.
  %L = {(a, i)} ∪ %rest ⋆ a ∉ proj1(%rest) ⋆ $OptThree(; %rest) ⋆ i ∈ {1} ∪ {2} ∪ {3};
  {
  ghost 'unfold $OptThree(; ?)';
  if a = a3 then {
    skip
  } else {
    ghost 'lemma unfold_optThree()';
    ghost 'fold $OptThree(; %L − {(a, i)})'
  }
  }
```

Figure 2.14: Example of a lemma for unfolding a pure list predicate at a particular element

proceed past the recursive call, there must be an instance of $OptThree for the same tail that was produced from the unfolding of $L. Naively one may think that a straightforward unfold is needed, however there is nothing to ensure that the unfolding will divide the set $\%L$ into the correct subsets. This is because the actual list has a structure and order enforced by the pointers on the heap. However, the pure predicates only define a set, with no ordering, so an unfold could non-deterministically choose *any element*. To overcome this a lemma *unfold_optThree* is used which will ensure the set is unfolded by the desired element. After the list has been incremented, a new instance of $OptThree must be introduced which can be achieved with standard fold operations. For clarity, the fold is separated for the two cases of whether the value is simply incremented or set to 1.

The specification and implementation of the special unfold lemma is given in Figure 2.14. The pre-condition requires the list predicate, and that a particular element is a member. The post-condition is essentially the definition of $OptThree, using the same (a, i) from the pre-condition. The body unfolds the predicate and tests whether the unique part of the element that was actually unfolded, which will have been skolemized to a freshly named a (for example a_3), is equal to the sought element. If so, then the unfold has occurred at the correct point and no further proof is needed. If the currently unfolded element is not the correct one, the lemma is applied recursively to the tail to extract the correct element. It is then necessary to fold the wrong element that was originally unfolded, into the tail that would have been given by the recursive call.

Note that such lemmas can be given stronger or weaker specifications as desired, depending on what is known about the content of the list. For instance the membership

constraint in the pre-condition could be either:

1. $(a, i) \in \%L$ – as in Figure 2.14.
2. $\%L = \{(a, i)\} \cup \%rest$ – $\%rest$ becomes $\forall$-quantified over the entire spec., rather than existential in the post-condition.
3. $a \in \text{proj}_1(\%L)$ – i becomes $\exists$-quantified in the post-condition.

Other examples where these pure lists may be useful is for defining a predicate that describes a relation between two linked lists, for instance a before and after. This assertion could not be added to the linked list's definition.

2.6 SMT-solver integration for pure reasoning

In common with many other verification tools (e.g. [60, 61, 40, 62]), the pure reasoning for Crowfoot is delegated to an SMT-solver. Recall that the logic of Crowfoot uses the $\vdash_{SMT}$ judgement for the pure entailments. This section describes the steps required to encode entailment problems that are generated by Crowfoot into assertions in the language of the SMT solvers.

The task for utilizing a SMT-solver comes down to encoding an entailment problem from the verification tool used here, to the solver's language. The translation is not trivial because the solvers do not include theories for sets or strings, which are required here. As we need to prove entailment, judgements of the form $A \vdash_{SMT} B$ are passed to the SMT solvers as $\neg(A \wedge \neg B)$. This is the essence of a problem given to the SMT-solver, it remains to ensure that A and B are appropriately translated.

The encoding of assertions is now described, making special note of the interesting cases of strings and multi-dimensional sets.

2.6.1 Translation to language of SMT solvers

Strings There exists string solvers (e.g. [63, 64, 65]) which can handle string reasoning. An option would be to utilize such a tool here, and use it in conjunction with an SMT solver for arithmetic reasoning. However, because the language for assertions allows occurrences of string expressions in a variety of formulae including set elements and predicate arguments, this approach would be difficult to implement due to the mutual dependence. Furthermore such string solvers are able to support far more complex strings than needed here (for instance regular expression handling). Instead, thanks to the simple subset of string operations supported in Crowfoot, a basic encoding of strings can be devised for handling by the SMT solver. The approach to this uses the theory of bit-vectors, which is a built in theory of many SMT-solvers. The approach of using bit-vectors to represent strings is also used in some of the string constraint solvers [65].

The translation for constant strings is a simple binary encoding of character codes. To keep the implementation simple only a limited alphabet of strings is allowed, which is the 52 upper and lower case letters. By subtracting 64 from the ASCII codes, this will allocate

a number from 1 to 58, leaving 0 available for the empty string. This means characters can be represented in fixed vectors of 6 bits, which must be 0-padded if required. By this method concatenation and splitting a string into individual characters are trivial. The encoding function (bin : char $\rightarrow \{0,1\}^6$) will be used in the translation described at the end of this section.

The theory of bit vectors supported by SMT solvers requires fixed length bit-vectors, and binary operations such as equality will give the expected result only on two same-typed instances. For example, "abc" $\neq$ "abcd" must ensure that both strings are the same length when converted to bitvectors, which will entail padding the left-hand side with an additional six zeroes. Similarly if there are two predicate instances \$S(;;"abc") and \$S(;;"abcd"), when the predicate is defined in the SMT solver's context its parameters must have a fixed type, which will be the maximum length string over all occurrences. Therefore it is necessary to infer the length of all string variables and expressions in an assertion passed to the SMT solvers.

It is assumed that the SMT solvers include a type declaration bitvector(n) for the type of a n-length bit-vector, a constructor $mk\text{-}bv(\_)$, and a concatenation operator $bv\text{-}concat(\_,\_)$. A translation T is defined from Crowfoot assertions to SMT solver assertions. An instance of this translation would be the following:

$$T(@s = \text{"abc"} {+\!+} @r) \quad \rightarrow \quad @s = bv\text{-}concat(mk\text{-}bv(100001100010100011), @r)$$
$$\text{where } @r :: \text{bitvector}(6), \quad @s :: \text{bitvector}(24),$$
$$bin(\text{a}) = 100001, \quad bin(\text{b}) = 100010, \quad bin(\text{c}) = 100011$$

The above example includes equality and concatenation of strings. The Crowfoot formula being translated includes two string variables and a constant string. The type of $@r$ is not inferable from the given formula, and so can be approximated to be some non-empty length chosen as one character, which is 6 bits. The type of $@s$ *can* be inferred, as long as the type of $@r$ has been discovered. The 24 comes from the 6 bits for each of the three characters in the constant, plus the length of $@r$. The binary encodings of each of the character can be provided already concatenated.

The algorithm for identifying the length of all string expressions in an assertion involves annotating each expression and term with an integer length. When an expression (e.g. $@a = @b$) or compound term (e.g. $@a{+\!+}@b$) is found, the type of each sub-part must be ensured to be equal by using the maximum of each. Only when a constant string is found can an actual length be given. Otherwise it is safe to assume the length is an arbitrary number greater than zero. The type inference algorithm does no intelligent look-ahead or behind, and so it will take several iterations in order to fully identify compatible types for each expression. This algorithm takes place at the same type as the type inference for sets.

Sets Most integration of SMT solvers into formal proof systems includes support only for basic sets [66]. Here where there are sets-of-sets more work is obviously required, although

as will be seen, so long as the SMT solvers support higher-order functions the extension is straightforward. The types needed to support sets are the function type ($\rightarrow$), and a type of tuples. A constructor for tuples *mk-tuple*(_) is assumed. The following example shows the translation of a simple formula, which includes an instance of sets-of-sets where $\%R$ is nested inside an element of $\%S$:

$$T(\ (x, y, \%R) \in \%S\ ) \quad \rightarrow \quad (\%S(\textit{mk-tuple}(x, y, \%R)))$$
$$\text{where } x, y :: \mathbb{Z}, \quad \%R :: (\mathbb{Z} \rightarrow \mathbb{B}), \quad \%S :: ((\mathbb{Z} \times \mathbb{Z} \times (\mathbb{Z} \rightarrow \mathbb{B})) \rightarrow \mathbb{B})$$

There are four variables used in the Crowfoot formula, which must each be typed. The integer variables x and y are immediate. There is not enough information in the formula to ascertain the type of $\%R$, except to say that it is a set, so a default type of a set of integers is assumed. Due to sets being represented as functions to booleans, the type of $\%R$ is a function from integer to boolean. The type of $\%S$ can be inferred because there is an explicit element available, which is a tuple of integer, integer, set of integer. The type in the SMT context is therefore a function from a tuple which includes a function as one element, to boolean.

To achieve this, a translation T is defined according to Figure 2.15.

There is a map $\textit{VarMap}^*(\_)$ for each variable type (ordinary (int) variables, predicate variables, collection (set) variables, string variables), from Crowfoot variable names to variables in the SMT solver's context. Every variable must be defined for the SMT solver in advance. Predicate variables are uninterpreted functions from a tuple of their parameter types, to boolean. For example the predicate instance $\$\mathrm{P}(x, y)$ will require the definition of a function $\$\mathrm{P} : (\mathbb{Z} \times \mathbb{Z}) \rightarrow \mathbb{B}$.

Set expressions are translated to functions from the type of elements to boolean. Set variables α are uninterpreted functions. For other set expressions, they are defined by a lambda binding. The projection function is defined by introducing existential quantifiers for all parts of a tuple except the sought index.

String expression translation involves the use of bit-vector constructor *mk-bv*(_) and concatenation function *bv-concat*, and the encoding function *bin* of characters to a 6bit binary described in the previous subsection.

Note that the constraints obtained from the translation of points-to atomic formulae are already present for entailments, due to a series of preparation phases performed by the verification system which adds facts that all addresses are distinct and non-zero. This takes place before an entailment problem will reach the SMT solver. It is important because a points-to formula such as $a \mapsto \_$ may be removed by cancellation, which would lose the implicit pure facts that may be important for completing the entailment, including $a \neq 0$. However it is necessary to include them here because the SMT solver is also used to check inconsistency of symbolic heaps (for instance after each symbolic execution step), for which no such "preparation" stages exist. There are two functions used: *distinct*(...) gives pairwise distinctness, and *ptAddrs*(...) yields the set of all address expressions used on the left-hand side of a points-to ($\mapsto$) operator.

Conjunctions:

$T(A_1 \star \ldots \star A_n) \quad \rightarrow \quad T(A_1) \wedge \ldots \wedge T(A_n) \wedge \mathit{distinct}(\mathit{ptAddrs}(A_1 \star \ldots \star A_n))$

Atomic formula:

$T(e_A \mapsto C) \quad \rightarrow \quad \neg(e_A = 0)$

$T(\mathrm{P}(e_V^*; e_S^*)) \quad \rightarrow \quad \mathit{VarMap}^{pvar}(P)(T(e_{V_1}), \ldots, T(e_{Vm}), T(e_{S_1}), \ldots, T(e_{Sn}))$

$T(e_{V_1} = e_{V_2}) \quad \rightarrow \quad T(e_{V_1}) = T(e_{V_2})$

$T(e_{V_1} \neq e_{V_2}) \quad \rightarrow \quad \neg(T(e_{V_1}) = T(e_{V_2}))$

$T(e_E \in e_S) \quad \rightarrow \quad T(e_S)\ T(e_E)$

$T(e_E \notin e_S) \quad \rightarrow \quad \neg(T(e_S)\ T(e_E))$

$T(e_{S_1} \subseteq e_{S_2}) \quad \rightarrow \quad \forall x.\ T(e_{S_1})\ x \Rightarrow T(e_{S_2})\ x$

$T(e_{S_1} = e_{S_2}) \quad \rightarrow \quad \forall x.\ T(e_{S_1})\ x \Leftrightarrow T(e_{S_2})\ x$

$T(e_{S_1} \neq e_{S_2}) \quad \rightarrow \quad \neg(\forall x.\ T(e_{S_1})\ x \Leftrightarrow T(e_{S_2})\ x)$

$T(e_{T_1} = e_{T_2}) \quad \rightarrow \quad T(e_{T_1}) = T(e_{T_2})$

$T(e_{T_1} \neq e_{T_2}) \quad \rightarrow \quad \neg(T(e_{T_1}) = T(e_{T_2}))$

Set expressions:

$T(\alpha) \quad \rightarrow \quad \mathit{VarMap}^{cvar}(\alpha)$

$T(\mathrm{proj}_i(e_S)) \quad \rightarrow \quad \lambda x.\ \exists e'_1, \ldots, e'_n.T(e_S)(e'_1, \ldots, e'_{i-1}, x, e'_{i+1}, \ldots, e'_n)$
where $e_S : t_1 \times \ldots \times t_n$

$T(e_{S_1} \cup e_{S_2}) \quad \rightarrow \quad \lambda x.\ (T(e_{S_1})x) \vee (T(e_{S_2})x)$

$T(e_{S_1} \setminus e_{S_2}) \quad \rightarrow \quad \lambda x.\ (T(e_{S_1})x) \wedge (\neg(T(e_{S_2})x))$

$T(\{e_E\}) \quad \rightarrow \quad \lambda x.\ x = T(e_E)$

$T(\emptyset) \quad \rightarrow \quad \lambda x.\ \mathit{false}$

Element expressions:

$T(e_V) \quad \rightarrow \quad T(e_V)$

$T(e_T) \quad \rightarrow \quad T(e_T)$

$T((e_{E1}, \ldots, e_{E2})) \quad \rightarrow \quad (T(e_{E1}), \ldots, T(e_{E2}))$

$T(e_S) \quad \rightarrow \quad T(e_S)$

Value expressions:

$T(n) \quad \rightarrow \quad n$

$T(x) \quad \rightarrow \quad \mathit{VarMap}^{ovar}(x)$

$T(e_{V_1} \oplus e_{V_2}) \quad \rightarrow \quad T(e_{V_1}) \oplus T(e_{V_2})$

String expressions:

$T(\text{“}S_1 \ldots S_n\text{”}) \quad \rightarrow \quad \mathit{mk\text{-}bv}(\mathit{bin}(S_1) \ldots \mathit{bin}(S_n))$

$T(s) \quad \rightarrow \quad \mathit{VarMap}^{svar}(@s)$

$T(e_{T_1} + + e_{T_2}) \quad \rightarrow \quad \mathit{bv\text{-}concat}(T(e_{T_1}), T(e_{T_2}))$

$$\mathit{distinct}(x_1, \ldots, x_n) = \forall 0 < i < j < n.\ \neg(x_i = x_j)$$

$$\mathit{ptAddrs}(\Phi) = \{e_A \mid e_A \mapsto \mathscr{C} \in \Phi\}$$

Figure 2.15: Transformation from assertions to SMT constraints

Step	Entailment problem	Proved by Yices
1	$\forall z.\ (\lambda a.a = (1,x))(z) \Leftrightarrow (\lambda a.a = (1,y))(z) \quad \vdash \quad x = y$	FAIL
2	$\forall z.\ (z = (1,x)) \Leftrightarrow (z = (1,y)) \quad \vdash \quad x = y$	FAIL
3	$\forall z.\ (1,x) = (1,y) \quad \vdash \quad x = y$	OK
4	$(1,x) = (1,y) \quad \vdash \quad x = y$	OK

Table 2.1: Example showing the limitations of Yices ($A \vdash B$ encoded as $\neg(A \wedge (\neg B))$)

2.6.2 Choice of SMT solver

Initially, the solver used was Yices [56]. This was chosen due to the multi-platform support and API availability[2]. Eventually, however, Yices proved insufficient for proving the more complex problems generated by Crowfoot where sets of tuples are involved.

An example of a weakness point of the Yices solver is given in Table 2.1. The entailment problems represent the encoding of

$$\{(1,x)\} = \{(1,y)\} \quad \vdash \quad x = y \tag{2.19}$$

into the SMT logic, with a reduction taking place at each step. Whilst the first two are not particularly complicated they do involve the $\forall$-quantifier, with quantifiers known to be a weakness of the Yices solver.

As a result of the limitations reached with Yices, Z3 [55] was chosen as a second solver based on its strong performance at the SMT competition *SMT-COMP*[3], and it proved more capable of providing answers for the type of entailment problems generated by the verification system. This includes the ability to prove the above example.

The principles of the encoding from the Crowfoot language to the SMT solver's respective languages are essentially the same, regardless of the actual solver, with the minor difference being that Yices's syntax supports uninterpreted functions, whereas the same results are achieved in Z3 through arrays. So a function application in Yices is used where an array select is for Z3. Internally the solvers reduce arrays to uninterpreted functions anyway, so it makes no difference [67]. There is a standardized language (SMT-LIB), supported by most solvers, which would remove the differences completely. However, to take advantage of APIs and special features and possibly more efficient implementations for the "native" input languages, this was not used.

[2] API used with "Ocamlyices" OCaml binding for Yices, Mickaël Delahaye, https://github.com/polazarus/ocamlyices

[3] SMT-COMP 2011 results, comparing Z3 with an earlier version of Yices, at: http://www.smtcomp.org/2011/

2.7 Further prover hints and efficiency considerations

2.7.1 Heuristics for instantiation of ∃-variables appearing in tuples

In Section 2.1, it was discussed how lemmas can be be used to assist the SMT solvers in deciding entailments. Such efforts can improve efficiency of the automated verification process because the SMT solvers are able to spend less time. However there is another layer at which efficiency can be improved, which is before entailments even need to be sent to the SMT solver.

The occasion arises when one is presented with an entailment that includes existentially quantified variables on the right-hand side. In order to show the entailment, obviously these variables need to be instantiated with witnesses from the left. Initially the verification system "cancels out" spatial formulae ($\mapsto$ and predicates), using a set of rules which assist the instantiation. These rules can be used for instance to match predicate arguments or contents of cells at the same address:

$$\begin{array}{lcl} e \mapsto x \ \vdash \ \exists v.\, e \mapsto v & \rightsquigarrow & e \mapsto x \ \vdash \ e \mapsto x \\ \mathrm{P}(x,y) \ \vdash \ \exists v.\, \mathrm{P}(x,v) & \rightsquigarrow & \mathrm{P}(x,y) \ \vdash \ \mathrm{P}(x,y) \end{array}$$

However, once the entailment involves a pure right-hand side, and all arguments of pure predicates have been instantiated, then there was no intelligent method for choosing suitable instantiations. This is an issue because the tool would simply guess instantiations by choosing from all the variables on the left-hand side. When the left-hand side is large and contains a number of different variable names, iterating through the possible combinations is inefficient.

A solution to alleviate some of this issue is to use more rules for the entailment prover that first choose variables that are *more likely* to be the ones needed to lead to a successful entailment. Two rules that have proved useful are the following.

$$\textsc{InstTuplePosition}\quad \frac{\Phi \ \vdash^{I} \ \exists \vec{v}\,.\,\Upsilon[v\backslash e_{E_i}] \star \Theta}{\Phi \ \vdash^{I[v:=e_{E_i}]} \ \exists \vec{v}, v.\ \Upsilon \star \Theta} \quad \begin{array}{l} (e'_{E_1}, \ldots, e'_{E_n}) \in \mathit{eltExps}(\Upsilon) \\ (e_{E_1}, \ldots, e_{E_n}) \in \mathit{eltExps}(\Phi) \\ e'_{E_i} = v \\ \textbf{backtracks} \end{array}$$

$$\textsc{InstSkolemName}\quad \frac{\Phi \ \vdash^{I} \ \exists \vec{v}\,.\,\Upsilon[v\backslash x] \star \Theta}{\Phi \ \vdash^{I[v:=x]} \ \exists \vec{v}, v.\ \Upsilon \star \Theta} \quad \begin{array}{l} x \in \mathit{filterSkolemsByName}(v, \mathit{fv}(\Phi)) \\ \textbf{backtracks} \end{array}$$

The first, (InstTuplePosition), instantiates variables that appear in a tuple on the right-hand side with expressions that appear at the same position in same-length tuples on the left. For instance

$$\{(a,b,c)\} = \alpha \ \vdash \ \exists v.\ (x,v,z) \in \beta \quad \rightsquigarrow \quad \{(a,b,c)\} = \alpha \ \vdash \ (x,b,z) \in \beta$$

. The function *eltExps* is used to filter all element expressions (e_E) from an assertion.

The second rule makes use of concrete variable names. The work is done by the function in the side-condition, which filters the set of all variables on the left-hand side to leave only those whose name is a skolemized form of the first argument. In terms of regular expressions, the function will return all variables in $fv(\Phi)$ that match the expression `/v[0-9]*/`, for given name v. This rule helps because of the way in which existential variables are skolemized by giving them a fresh name by appending a fresh integer. As an example, consider a predicate (recdef $\$\mathrm{P}(x) := \exists y.\ x = y$). When this predicate is unfolded, the existential y will be skolemized to a fresh y_n, for the next unused integer n. If the predicate then needs to be folded back up, then it can do so by instantiating the y by the introduced skolemized variable. This is demonstrated by this annotated snippet, which shows that y should be instantiated with the skolem variable y_4:

State 1:$\{\Phi \star \$\mathrm{P}(a)\}$
ghost 'unfold $\$\mathrm{P}(a)$';
State 2:$\{\Phi \star x = y_4\}$
ghost 'fold $\$\mathrm{P}(a)$'; // r.t.p.: $\Phi \star x = y_4 \vdash \exists y.\ x = y$

There is scope for more such rules which may emerge as useful after patterns start appearing in more specifications and examples.

2.7.1.1 Soundness

InstTuplePosition By the definition of $\vdash^I$, it can be assumed that $fv(\phi) \cap \vec{v}, v = \emptyset$. It must first be proved that $fv(\Phi) \cap \vec{v} = \emptyset$, which follows from the above assumption. By the soundess of $\vdash^I$, it can be assumed:

(a') $\Phi \Rightarrow \Upsilon[v \backslash e_{E_i}][\vec{v} \backslash I(\vec{v})] \star \Theta$
(b') $fv(\Theta) \subseteq fv(\Phi)$
(c') $\mathsf{dom}(I) = \vec{v}$

It must be proved that:

(a) $\Phi \Rightarrow \Upsilon[v \backslash e_{E_i}][\vec{v}, v \backslash I[v := e_{E_i}](\vec{v}, v)] \star \Theta$
(b) $fv(\Theta) \subseteq fv(\Phi)$
(c) $\mathsf{dom}(I[v := e_{E_i}]) = \vec{v}, v$

(b) and (c) follow from (b') and (c'). It can be shown that (a) follows easily from (a'), given that the side condition implies that $e_{E_i} \in fv(\Phi)$ and the assumption $fv(\phi) \cap \vec{v}, v = \emptyset$:

Φ
$\Rightarrow$ {By (a')}
$\Upsilon[v \backslash e_{E_i}][\vec{v} \backslash I(\vec{v})] \star \Theta$
$\Leftrightarrow$ {By fact that $fv(e_{E_i}) \cap v = \emptyset$ (by side condition + assumption)}
$\Upsilon[v \backslash e_{E_i}][\vec{v}, v \backslash I[v := e_{E_i}](\vec{v}, v)] \star \Theta$

```
const a;
recdef $Spec := a ↦ _ ⋆ x ↦ _;
proc main(f, g)
   pre : f ↦ _ ⋆ g ↦ _;
   post : f ↦ ∀x. {$Spec(x)} · (x) {$Spec(x)} ⋆ g ↦ ∀x. {$Spec(x)} · (x) {$Spec(x)} ;
{
   [f] := foo(_);
   [g] := bar(_) "deepframe a ↦ _
                  pre fold $Spec(?)
                  post fold $Spec(?)"
}
```

```
proc foo(x)
   pre : $Spec(x);
   post : $Spec(x);
{...}
```

```
proc bar(x)
   pre : x ↦ _;
   post : x ↦ _;
{...}
```

Figure 2.16: Example demonstrating the use of folding during the storing of a procedure

InstSkolemName The soundness of the second instantiation rule is a corollary of the first rule, substituting e_{E_i} with x in the proof.

2.7.2 Hints for folding specifications during storecode

There are some cases where it may be useful to consider a number of procedures on the heap as fitting the same, strongest, behavioural specification. However, one of the advantages of the frame rule (and rule of consistency) is that one can have a level of modularity whereby a specification of a procedure does not need to mention parts of the heap that are unaffected. For convenience, and to avoid lengthy duplication of assertions, it can be desirable to use a predicate to represent some generic pre- and/or post-condition that fits a number of procedures. For the procedures that require the whole specification, their specification can be precisely that predicate. But for those that only require a smaller version, one needs a technique for applying the deepframe rule and afterwards performing a fold operation on the resulting assertions.

To make this explanation clear, consider the simple, contrived example in Figure2.16. The post-condition of *main* shows the desired result where two procedures fulfil identical specifications. However, the body shows that f and g refer to two different procedures, which have different specifications. The solution is the hint to the second store-code operation which instructs the prover to first use the deepframe rule to add an invariant (in this case a cell pointed to by constant a), and then perform a fold on both the pre-condition

and post-condition.

To support these annotations, a special version of the rule for store-code statements was created. This rule already appears in [52], although was developed as part of the work in this thesis. The new rule is in Figure 2.17. It makes use of a new $\vdash^{I}_{G}$ judgement, which is still essentially $\Rightarrow$, however the additional G annotation instructs that some predicate must be folded. The two rules defining the new judgement are (FoldPredRight) and (FoldPredLeft), which allow the folding in either direction. Note that the $\star$ emp in the (FoldPredRight) case is to ensure that there are no leftover formula, as a frame may have been produced by the $\vdash^{I}$ judgement which is required in other usages.

StoreCodeGuided

$$\frac{\begin{array}{c}\Pi : ((\Phi \otimes \exists \vec{a}.\Upsilon') \star \Upsilon'[\vec{a}\backslash\vec{a'}])[\vec{v}\backslash\vec{v'}] \;\vdash^{\emptyset}_{G^{\text{pre}}}\; X(\vec{d}) \star \Phi' \\ \Pi : ((\Theta \otimes \exists \vec{a}.\Upsilon') \star \Upsilon'[\vec{a}\backslash\vec{a'}])[\vec{w}\backslash\vec{w'}] \star \mathsf{emp} \;\dashv\vdash^{\emptyset}_{G^{\text{post}}}\; Y(\vec{e}) \star \Theta' \\ S = \left(\begin{array}{c} \left\{ \; \exists \vec{v}, \vec{a'}.\ (X(\vec{d})[\vec{v}\backslash\vec{v''}] \star \Phi')[\vec{v'}\backslash\vec{v}] \; \right\} \\ \forall \vec{p}|_U, \vec{x}. \qquad \cdot(\vec{p}|_U) \\ \left\{ \; \exists \vec{w}, \vec{a'}.\ (Y(\vec{e})[\vec{w}\backslash\vec{w''}] \star \Theta')[\vec{w'}\backslash\vec{w}] \; \right\} \end{array} \right)[\vec{p}|_{I\backslash U}\backslash\vec{t}|_{I\backslash U}] \\ \mathit{purify}(\Upsilon) \vdash_{SMT} E = E' + o \\ \Pi; \Gamma, \{\exists \vec{v}.\Phi\}\, \mathcal{F}(\vec{p})\, \{\exists \vec{w}.\Theta\} \triangleright \left\{ \Upsilon \star E' \mapsto \mathscr{C}_0, \ldots, \mathscr{C}_{o-1}, S, \mathscr{C}_{o+1}, \ldots, \mathscr{C}_n \right\} C \left\{ Q' \right\} \end{array}}{\Pi; \Gamma, \{\exists \vec{v}.\Phi\}\, \mathcal{F}(\vec{p})\, \{\exists \vec{w}.\Theta\} \triangleright \left\{ \; \Upsilon \star E' \mapsto \mathscr{C}_0, \ldots, \mathscr{C}_n \; \right\} \begin{array}{l} [E] := \mathcal{F}(\vec{t}) \\ \text{`deepframe } \exists \vec{a}.\ \Upsilon' \\ \text{pre } G^{\text{pre}} \\ \text{post } G^{\text{post}} \text{'}; \; C \end{array} \left\{ \; Q' \; \right\}}$$

where $|\vec{t}| = |\vec{p}|$ and t_i either value expression a_i or _;
$\vec{x} = fv(S\text{-precondition}, S\text{-postcondition}) - \vec{p}$; $\vec{v'}, \vec{w'}, \vec{a'}, \vec{v''}, \vec{w''}$ fresh
$\vec{p} = (p_i)_{i \in I}$; $U = \{i \in I \mid t_i = \_\}$ $\vec{p}|_X = (p_i)_{i \in I \cap X}$

FoldPredRight

$$\frac{(\mathrm{X}(\vec{v}) \Leftrightarrow (\exists \vec{v}_1.\ \Upsilon_1) \vee \ldots \vee (\exists \vec{v}_n.\ \Upsilon_n)) \;\in\; \Pi \qquad \bigwedge_{1 \le i \le n} (\Upsilon_i[\vec{v}\backslash\vec{e}][\vec{v_i}\backslash\vec{d_i}] \star \Theta \;\vdash^{\emptyset}\; \Phi \star \mathsf{emp})}{\Pi : X(\vec{e}) \star \Theta \;\vdash^{\emptyset}_{G}\; \Phi \star \mathsf{emp}} \quad \begin{array}{l} \vec{d_1} \ldots \vec{d_n} \text{ fresh} \\ G = \mathsf{fold}\ X(\_)\ \_ \end{array}$$

FoldPredLeft

$$\frac{\mathrm{X}(\vec{v}) \Leftrightarrow (\exists \vec{v}_1, \vec{b}_1.\ \Upsilon_1) \vee \ldots \vee (\exists \vec{v}_n, \vec{b}_n.\ \Upsilon_n) \;\in\; \Pi \qquad \Phi \;\vdash^{I}\; \exists \vec{w}, \vec{c}.\ \Upsilon_i[\vec{v}\backslash\vec{q}][\vec{b}\backslash\vec{y}][\vec{v_i}\backslash\vec{w}] \star \Theta}{\Pi : \Phi \;\vdash^{\emptyset}_{G}\; X(\vec{e}) \star \Theta}$$

$G = \mathsf{fold}\ X(\vec{p})\ \vec{b} = \vec{y}$,
$\vec{b_i} \subseteq \vec{b}$, $\vec{b} - \vec{b_i} \cap (fv(\Upsilon_i) \cup \vec{p}) = \emptyset$
$1 \le i \le n$, $\vec{c}, \vec{w}$ fresh,
each p_i is an expression or ?,

$$\text{each } q_i \text{ is } \begin{cases} c_i & \text{if } p_i \text{ is ?} \\ p_i & \text{otherwise} \end{cases},$$

$$\text{each } e_i \text{ is } \begin{cases} I(c_i) & \text{if } p_i \text{ is ?} \\ p_i & \text{otherwise} \end{cases}$$

Figure 2.17: Additional symbolic execution and entailment rules for store-code hints

The store-code rule is complicated in presentation by the necessary freshening of quantified variables, however its intuitive meaning is fairly simple. The first and second premises perform the folds from the G^{pre} and G^{post} annotations by applying them to the pre- and post-condition of the specification of $\mathcal{F}$ which comes from the procedure context. The third premise uses the resulting assertions to construct a new specification S for the procedure being stored, which also handles the partial application where some parameters in $\vec{t}$ may be provided. The fourth premise calculates the address and offset for E, and the last premise updates the content with the new specification S. The soundness of these three rules is included in [52].

2.8 Discussion

This chapter has detailed a number of extensions to the original Crowfoot system. The extensions were each implemented on a by-need bases due to the requirements encountered as examples became more complicated. As such, these extensions are by no means exhaustive in improving the capabilities or efficiencies of the system and it is expected that more may be helpful for future examples. These extensions have, however, demonstrated that the system is sufficiently modular that enhancements can be added non-invasively without major alterations to the underlying logic or implementation.

In terms of the specific extensions here there are several elements to discuss. Firstly, the introduction of lemmas. These were inspired by the VeriFast tool [42], which also interprets them essentially as procedures with restricted syntax for commands. A benefit of VeriFast's implementation is that it checks termination when lemmas make recursive calls using inductive predicates. They do this by one of three methods, which could all conceivably be implemented in Crowfoot. A further benefit, although not directly applicable here, is that they allow *lemma function pointers* which has applications in verification of concurrent programs where auxiliary variables are introduced to track individual threads' effect on a shared resource. A benefit of the Crowfoot implementation of lemmas is that there is greater flexibility given the provision for *abstract* lemmas, which allow one to specify properties that are outside the scope of the verifier's abilities.

A consideration that arises from the need for lemmas in the first place is that they are sometimes used to prove inductive properties. Work on the Dafny [61] verifier has led to work in harnessing the power of SMT solvers to perform induction proofs [68]. Although Dafny's language allows the use of universal quantification, whereas inductive properties in Crowfoot would need to be specified by predicates, it would be beneficial to make use of the method for proving induction with SMT so that certain lemmas can be proved automatically (with body *skip*).

Other related work in this respect, such as CFML [69] or Why3 [70], makes use of automated theorem provers at the point where proofs get stuck. This interactive approach has the benefit that theorem provers are able to perform inductive reasoning such that some lemmas may also be avoidable. On the subject of interactivity, it would be helpful

for the user performing verification if annotations could be provided on-the-fly during verification at each point where the proofs cannot proceed, which does not require use of a theorem prover. This is possible in VeriFast [42]. The other annotations that are available in Crowfoot are variable instantiation hints, which again would benefit from an interactive intervention during the proof. This is particularly the case for providing hints for variables that need instantiating with skolemized variables. At the initial stage of annotating a program with specifications and additional hints, it will probably not be immediately obvious what integer has been used to freshen some variable (e.g. a_2 or a_6) although it is deterministic. It is often the case that it is necessary to attempt the verification, inspect the symbolic state at the point where the problem is, go back to the source code and add/update the instantiation hints with the discovered variable name, and finally verify properly. Crowfoot does allow the use of a question mark in place of an unknown skolem number ($a?$), which will choose a variable based on the name before the integer, however this will not be reliable in cases where there are two skolemized versions of the same original variable name.

The type extensions to the programming and assertion language included that of strings and sets-of-sets. It is worth noting that the main body of work in undertaking such language additions falls to finding appropriate encodings to the SMT solver language. The scope for introducing more language features therefore would seem to be limited by the capabilities of the SMT solvers. That said, however, there is certainly still the possibility to add further types, with some even directly supported by the SMT languages, for example arrays.

A bit-vector encoding of strings was naively chosen because of the natural representation of characters as bytes, and the ease of using the universal operations common to both SMT solvers. This does involve the extra cost of ensuring that the sizes of comparable bit-vectors are equal. A more intuitive encoding might be to use arrays (or functions in Yices which does not have explicit array types), however this would introduce more quantifiers in order to define constant strings as uninterpreted functions. The changes to Crowfoot would be trivial to implement as future work.

This chapter has included the first attempt of providing support for the Pottier's anti-frame rule [21] in an (semi)automated tool. The soundness of the new rule proved elsewhere [24] has been adapted to the procedural language here, and an algorithm for "subtracting" an invariant developed. Presently the distribution of the $\otimes$ operator is limited in the implementation where predicates are concerned and it would be beneficial to generalize this such that the restrictions on predicates being left-zeroes can be relaxed. This could be achieved by allowing recursive predicates to be declared using $\otimes$. Completing this extension to improve expressivity would have benefits with both the deep frame and anti-frame rules.

In terms of the underlying logic of Crowfoot, there is research to be done into the extent to which the results achieved with nested-Hoare triples and the deep frame rule can also be achieved with abstract predicates [37]. The "invariant" could be represented by an abstract predicate which can be instantiated as necessary for concrete examples.

The obvious advantage of the deep frame rule is that the invariant need not be included in specifications of the original code, as it is introduced only at the necessary points in proofs. Substituting for abstract predicates would mean a predicate would need to be added to the specification of any code where it *might* be needed, even if certain examples do not require it.

In supporting pure forms of lists with inductive predicates, no built-in support was included in the tool. There is a mechanism for splitting and joining heap linked list segments, which could be adapted to this case. The disadvantage with the present approach that lacks split/join annotations is that it requires providing (and proving) a lemma for every inductively defined pure predicate that will need to be split. The predicates will fit the pattern described in Section 2.5, so it seems sensible to have the built-in support. It should also be acknowledged that there is duplication of the definition of such predicates, because the post-condition of the lemma that does the splitting will also use it. To add built-in support for unfolding on an element should be a trivial adaptation of the existing implementation for linked list segments.

By the introduction of these pure lists, this thesis has explored further the types of inductive lists that have proved useful in specifying example programs in Crowfoot. This is by no means exhaustive, and the prover could be extended to include built-in support for generalised inductive data structures. However, the freedom of user-defined predicates already allows for a broader range of inductive types, not just lists, while lemmas can be provided to describe pertinent properties.

The use of SMT solvers has become a common practice in the implementation of verification tools, however there has been no decisive victor in which solver is the best weapon of choice. As each solver improves in efficiency and ability, it is possible that a better solver will be more appropriate for checking the pure entailments produced here. It is helpful to consult the annual results of the SMT-COMP[4] competition, however not all developers enter and the categories are organised based on categories of "logics". Here, a number of logics are required, primarily for handling integer arithmetic, arrays, and bit vectors. The advantages of SMT solvers though is that they are mostly compliant with a common standard, and to utilize a new solver would be a case of adapting the encoding to a new interface.

The SMT encoding for sets-of-sets uses only a simple type inference system, which does not support recursive types. Recursively typed sets can be useful for specifying generic data structures, such as XML. This usage is evidenced by the second case study in Chapter 4, which avoids the limitation by using a concrete structure. The SMT solvers already support recursive data types, although it is not clear whether they will impact performance, and the recursive set type issue could be overcome by implementing an appropriate type inference algorithm.

Finally, regarding the set reasoning by the SMT solver, an alternative approach to handling sets would be to define a new theory *in the solver* with the required set operations

[4] www.smtcomp.org

that can then be directly used in encoded assertions. This would allow for more of a straightforward encoding between the Crowfoot assertion language and the SMT solver's. Such an approach would transfer the burden of defining the properties of set operations to the SMT solver, by generalised axioms, rather than being declared for each individual formula as presently takes place in Crowfoot. Therefore the duplication of the encoded meaning for each set operation would be removed.

Chapter 3

Reasoning for reflective programs

Reflective programming allows a program to inspect and manipulate itself, at runtime. This runtime nature naturally restricts static verification, and as such techniques for allowing such static verification are weak.

Whilst it must be acknowledged that there will always be limitations to a technique providing static verification, it is not without its uses. The focus of this work is on a specific application of reflection, that of "closed" systems. One aspect of reflection allows loading new code at runtime, which can therefore be considered "open". In contrast, the closed programs to be supported here are those where the entire source code is known at verification-time. This is still useful because reflection can be used to create "generic" code snippets that can be plugged into different program contexts. Without the reflection, the same result would usually require the code to be highly dependent on the program's structure. As such, the solutions for creating generic/adaptable code are often more elegant when reflection is used. An example of this is the visitor pattern vs. a reflective version, discussed in Chapter. 4.

This chapter presents a method for providing a specified reflective library that may be used to verify programs that make use of the reflection. It focuses on supporting Java-style reflection, and includes specifications for a subset of Java's reflective library operations. The chapter begins by giving a description for metadata stored on the heap, making use of linked list structures. Following this the library procedures, which have been implemented using primitive heap manipulation, are given. These library procedures have been verified and specified. Finally it is described how the metadata can be automatically generated for a given input program.

3.1 Introduction

The first stage for supporting the reflection in the Crowfoot system is to devise a representation of the metadata, which is the description of a program's structure such as details of classes and methods. There are two options here:

1. Support the reflection by building it into the language with reflective operations as primitive commands, and create respective symbolic execution rules in the supporting logic. These rules then need to be proved sound.
2. Use the already available support for heap data structures and store the metadata on the heap such that the reflective operations can be implemented using the present, sound, heap-manipulation features.

The advantages of the first approach are that the metadata need not be exposed to the programmer/program, which prevents possible corruption. However the obvious disadvantage is that there is the task of developing new proof rules and proving their soundness. The second approach primarily utilises the proof rules that are already provided, and the reflective library can be *implemented*. This allows the correctness of the library to be asserted through the conventional verification. It is this second approach that has been undertaken here.

The reflection in this context will be supporting a simplified class-based object oriented programming paradigm, similar in style to Java in terms of the available reflective methods[1]. One can consider that an example written in a high-level language with reflection, such as Java, can be systematically translated into the Crowfoot language for verification. The appropriate classes used by reflection in Java, known as metaclasses, which are adopted here are: Object, Class, Method, Field, and Constructor.

A simple representation of objects is presented in Section 3.2. The representation of metadata on the heap is presented in Section 3.3. The reflective library that manipulates the metadata, which is implemented, specified and verified, is then given in Section 3.4. The process of generating the metadata from the program's source code is given in Section 3.5, and soundness of all the reflection aspects is shown in Section 3.6.

3.2 A simplified class/object representation

As mentioned in Section 3.1, the supported reflective features are based on those available in Java. This means that, when translating a Java-like program into the procedural language used here, a representation of classes and objects is required. This representation is simplified at this stage, discarding properties such as inheritance and security/visibility properties. The representation of objects on the heap is described now, with the details of classes presented with the metadata structures in Section 3.3.

For an object held on the heap, there is a record of the (dynamic) class type, and a pair of cells for each field of the object. A concrete example object held at address o with three fields, would look like:

$$o \mapsto @clsA,\ 1, f1,\ 1, f2,\ 1, f3,\ 0$$

where $@clsA$ is a variable containing the name of some class. While not accessed by the programmer, for the reflection implementation the class of the object needs to be accessible

[1] http://docs.oracle.com/javase/6/docs/api/java/lang/reflect/package-summary.html

$$\begin{array}{l}\mathsf{recdef}\ \$\mathrm{Object}(ptr; \%fs; @class) := \\ \quad ptr \mapsto @class \star \$\mathrm{ObjFields}(ptr + 1; \%fs);\end{array}$$

$$\begin{array}{l}\mathsf{recdef}\ \$\mathrm{ObjFields}(ptr; \%fs) := ptr \mapsto 0 \star \%fs = \emptyset \\ \qquad \vee \\ \quad \exists\, val, @type, \%rest.\ ptr \mapsto 1, val \star \$\mathrm{ObjFields}(ptr + 2; \%rest) \\ \quad \star\ \ \%fs = \{(ptr, @type, val)\} \cup \%rest \star ptr \notin \mathrm{proj}_1(\%rest);\end{array}$$

Figure 3.1: Representation of objects on the heap, described by predicates

from the program (for instance, by the implementation of getClass), so that is included in the first cell. To generalize to having a varying number of fields for each object, recursively defined predicates are used to describe such an object on the heap. Each field is actually a pair of adjacent cells: the first cell is either 1 if there is another field, or 0 if there are no more fields (the end of the list). The second cell is the field's value. This enables the sequences of adjacent cells to be traversed, and means that field offsets are incremented by 2 each time.

One might think that this would be better represented by a standard linked list structure. This would work well for accessing fields via reflection, where the list can be traversed in the usual way. However, it is naturally desirable for object fields to also be accessible directly for known objects. That is, for example, $a = o.foo$ in Java. The obvious translation for such a statement in the Crowfoot language would be $a := [o + foo]$, where o is a pointer to an object, and foo is an address offset that may be a constant that can be used for all objects of the same class. Such an elegant translation would not be possible with a linked list, where it would be necessary to traverse the list to perform a simple field access.

The definitions of these predicates are given in Figure 3.1. The first predicate is the top-level predicate which encapsulates an object. The three arguments to this predicate are: the pointer to the object, a list of its fields, the class type. The $ObjFields predicate inductively describes a sequence of adjacent heap cells, enriched with a type of each field. The field types are not visible in the above example for the object at address o, and therefore the predicates allow for stronger assertions.

The above example object can thus be wrapped up into the following predicate instance

$$\$\mathrm{Object}(o; \{(o+1, @clsB, f1)\} \cup \{(o+3, @clsC, f2)\} \cup \{(o+5, @clsB, f3)\}; @clsA)$$

for some class names $(@clsB, @clsC)$ chosen for the field types.

Note that the heap structure representing objects used in an early version of a reflective visitor pattern example [58] used two sets to describe the fields, one for primitive integer fields, and one for fields containing proper objects. This can be avoided here by using a special keyword for the type of integers. The simple class name "`int`" is used, which

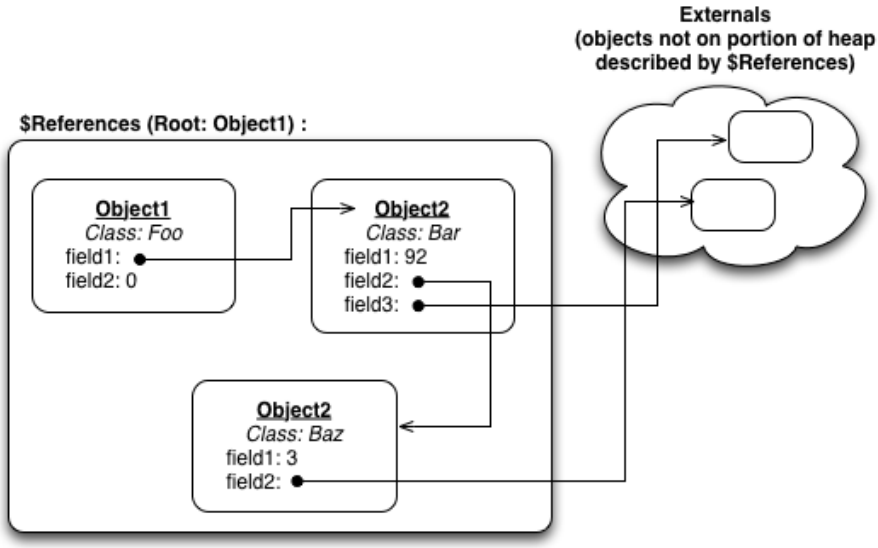


Figure 3.2: Graphical representation of a collection of objects

is not going to conflict with other user-declared classes because, as will be discussed in Section 3.5, recognised classes in a given program must start with an uppercase first letter.

A concept that will be useful later when considering how constructors initialize an object is to be able to wrap-up a connected collection of objects. That is, for a "root" object in the collection whose field references another object, that other object must also be on the heap, in the collection. In a sense, the goal is to describe the heap for the closure of an object and its references. The diagram in Figure 3.2 shows an example where the root object is *Object1*, which references two other objects (either directly or transitively). All three objects can be grouped together such that no field points to an object outside the collection. The definition of such a collection is complicated slightly by having the primitive (integer) fields and the object fields, because integer fields aren't referencing another object and so can be ignored when describing the rest of the heap from the context of an object.

The predicates for describing the "closure" of a collection of objects are defined in Figure 3.3. Predicate $FieldRefs provides the filtering for removing the primitive fields that won't be realised as objects on the heap. Essentially, the predicate models a function from a list of fields $\%F$ to a list of fields $\%R$ where $\%R$ are only those fields which are references (i.e. their class is not "`int`"). Its definition is pure, with induction over the first argument. The first disjunct declares that for an empty list of fields, the filtered result is also empty. In the second case, we see the current field is not primitive ($@type \neq$ "`int`"), so there is a constraint to ensure that its value appears in the result set. The same result set is passed through the tail predicate instance in order to allow for the same object to be referenced by multiple fields. The final case handles primitive fields, where no additional constraint appears.

The $\$\text{References}(;\%externals,\%R)$ predicate is heap-based, and describes a number of object predicate instances. Those in $\%R$ are on the heap wrapped up in the $References

$$
\begin{array}{l}
\mathsf{recdef}\ \$\mathrm{References}(;\%\mathit{externals},\%R) := \%R = \emptyset \\
\vee \\
\exists \mathit{ptr}, @\mathit{type}, \%\mathit{fs}, \%\mathit{rest}, \%\mathit{fieldRefs}. \\
\quad \$\mathrm{Object}(\mathit{ptr};\%\mathit{fs};@\mathit{type}) \star @\mathit{type} \neq \text{“}\texttt{int}\text{”} \star \%R = \{(\mathit{ptr}, @\mathit{type}, \%\mathit{fs})\} \cup \%\mathit{rest} \\
\quad \star\ \mathit{ptr} \notin \mathrm{proj}_1(\%\mathit{rest}) \star \mathrm{proj}_1(\%R) \cap \mathrm{proj}_1(\%\mathit{externals}) = \emptyset \\
\quad \star\ \$\mathrm{References}(;\%\mathit{externals} \cup \{(\mathit{ptr}, @\mathit{type}, \%\mathit{fs})\}, \%\mathit{rest}) \\
\quad \star\ \mathrm{proj}_1(\%\mathit{fieldRefs}) \subseteq \mathrm{proj}_1(\%R \cup \%\mathit{externals}) \star \$\mathrm{FieldRefs}(;\%\mathit{fs}, \%\mathit{fieldRefs});
\end{array}
$$

$$
\begin{array}{l}
\mathsf{recdef}\ \$\mathrm{FieldRefs}(;\%F,\%R) := \%F = \emptyset \\
\vee \\
\exists \mathit{ptr}, @\mathit{type}, \mathit{value}, \%\mathit{fs}, \%\mathit{rest}. \\
\quad \%F = \{(\mathit{ptr}, @\mathit{type}, \mathit{value})\} \cup \%\mathit{rest} \star \mathit{ptr} \notin \mathrm{proj}_1(\%\mathit{rest}) \star @\mathit{type} \neq \text{“}\texttt{int}\text{”} \\
\quad \star\ (\mathit{value}, @\mathit{type}, \%\mathit{fs}) \in \%R \star \$\mathrm{FieldRefs}(;\%\mathit{rest}, \%R) \star \$\mathrm{FieldRefs}(;\%\mathit{fs}, \%R) \\
\vee \\
\exists \mathit{ptr}, @\mathit{type}, \mathit{value}, \%\mathit{rest}. \\
\quad \%F = \{(\mathit{ptr}, @\mathit{type}, \mathit{value})\} \cup \%\mathit{rest} \star \mathit{ptr} \notin \mathrm{proj}_1(\%\mathit{rest}) \star @\mathit{type} = \text{“}\texttt{int}\text{”} \\
\quad \star\ \$\mathrm{FieldRefs}(;\%\mathit{rest}, \%R)
\end{array}
$$

Figure 3.3: Predicates defining a closure of set of objects on the heap

instance, and those in $\%\mathit{externals}$ are elsewhere on the heap, outside of the $References collection. For the non-empty case, the definition includes the current object, which must not be primitive. The implicit constraint that all object pointers are unique is explicitly included to reduce the number of predicate unfoldings needed to otherwise reveal the fact. The $\%\mathit{externals}$ argument is a list, of the same type as $\%R$, which is desired to contain any objects that are in a part of the heap not described by the predicate. The $\%\mathit{externals}$ set is necessary, otherwise objects could never be extracted because it would break the closure. Hence the inductive occurrence of $References adds the current object to this set. The last line ensures that all of the current object's fields which are references to objects must also be included in either $\%R$ or $\%\mathit{externals}$.

It is worth emphasizing that the $References predicate is primarily only used for a nullary constructor where a new object is created, and empty objects are created for each field. Ordinarily, a procedure does not need access to *all* fields of an object, in which case the $References predicate does not need to be utilised and instead a more specialized program-specific predicate can be more appropriate. The case studies in Chapter 4 show examples of both a case where a more concrete tree structure is suitable, and another where the $References is actually useful in a non-constructor setting.

$$\begin{aligned}&\mathsf{recdef}\ \$\mathrm{L}(a, z; \%L) \ := a = z \star \%L = \emptyset \\ &\vee \exists next, \%rest, \ \vec{V}\ . \\ &a \mapsto \mathscr{C}\ , next \star \ \Phi\ \star \$\mathrm{L}(next, z; \%rest) \star \%L = \{(a, \ \ldots\ )\} \cup \%rest\end{aligned}$$

Figure 3.4: Underlying linked-list definition used for metadata heap structures

3.3 Metadata representation

With the simple representation of objects in mind, the shape of the metadata can be described. The metadata consists of a description of the classes in the program, including methods, fields and constructors.

The principle of the metadata representation is fairly straightforward, using standard definitions of linked lists. However they are slightly more complex in that lists contain lists, and there are also some extra restrictions in place pertaining to the content of these lists.

The high-level description of the structures is:

- Metadata is a list of classes.
- Each class contains a name, a constructor, a list of methods and a list of fields.
- Each method contains a name, specification and a list of parameter types.
- Each field contains a name, type and a record of its declaring class.

To assist in reading the predicate definitions, the underlying linked list data structure common to each of the primary list predicates (classes, methods, fields) is given in Figure 3.4. Lists of this pattern are recognised by Crowfoot as splittable linked list segments. This structure is simply enriched to make stronger definitions in each concrete case. Predicate \$L defines a list from address a to z, with a "pure" representation of the lists elements in the set $\%L$. The first disjunct is the base case where the list is empty. In the second disjunct, the shaded parts are placeholders for variables and formulae specific to each concrete list example. Address a points to a list of adjacent heap cells which are the content of the list, and the last cell ($next$) being a pointer to the next element of the list. The rest of the list is therefore a list from $next$ to z. The set $\%L$ is a set (usually of tuples) allowing the contents of the list to be described in assertions. Typically a formula in Φ will declare some part of each element to be a unique key, for instance a class name is unique in the list of classes ($name \notin \mathrm{proj}_2(\%rest)$). In the concrete instances of this list definition pattern, the common parts will be in grey so the reader can focus on the individual content differences.

3.3.1 Classes

For the purposes of this work, the metadata for classes is restricted compared to Java, containing only a name, constructor, method list and field list. The list definition for class

$$
\begin{array}{l}
\textsf{recdef}\ \$\mathrm{ClassLseg}(a, z; \%cs) := a = z \star \%cs = \emptyset \\
\quad \vee \exists next, \%rest, @class, fs, \%fs, ms, construct, \%ms. \\
\quad a \mapsto @class, ms, fs, construct, next \\
\quad \star\ \$\mathrm{Constructor}(construct; \%fs; @class) \\
\quad \star\ @class \notin proj(2, \%rest) \\
\quad \star\ \$\mathrm{MethodLseg}(ms, 0; \%ms) \star \mathrm{proj}_3(\%ms) \subseteq \{@class\} \\
\quad \star \$\mathrm{FieldLseg}(fs, 0; \%fs) \star \mathrm{proj}_4(\%fs) \subseteq \{@class\} \\
\quad \star\ \$\mathrm{ClassLseg}(next, z; \%rest) \\
\quad \star\ \%cs = \{(a, @class, \%fs, \%ms, construct)\} \cup \%rest;
\end{array}
$$

$$
\begin{array}{l}
\textsf{recdef}\ \$\mathrm{Constructor}(ptr; \%cFs; @class) := ptr = 0 \star @class = \text{“}\texttt{int}\text{”} \\
\vee \\
\qquad\qquad \left\{\ res \mapsto \_\ \right\} \\
\qquad\qquad \cdot(res) \\
ptr \mapsto \forall res.\ \left\{ \begin{array}{l} \exists o, \%objFields, \%refs. \\ res \mapsto o \star \$\mathrm{References}(; \emptyset, \%refs) \\ \star\ (o, \%objFields, @class) \in \$\mathrm{refs} \\ \star\ \$\mathrm{FieldsExist}(; \%cFs, \%objFields) \end{array} \right\}
\end{array}
$$

Figure 3.5: Predicate describing class metadata on the heap

metadata is given in Figure 3.5, along with an auxilliary predicate describing constructors. The non-empty case shows that the heap contains the class name, a pointer to a method list, a pointer to a field list, a pointer to a constructor (discussed in a moment). The “pure” (heap-free) version of this is then what constitutes the tuple of the set, and the pointer a is needed because that is what is used for accessing the data. Note that there are three extra formulae, $@class \notin \mathrm{proj}_2(\%rest)$ ensuring that each class has a unique name, and $\mathrm{proj}_3(\%ms) \subseteq @class$ and $\mathrm{proj}_4(\%fs) \subseteq @class$ which links the “declaring class” part of the method and field metadata, respectively, (see Section 3.3.2 and 3.3.3) to the actual declaring class (i.e. every field in a class’s field list has its declaring class data equal to the class). The reason that these are set-inclusion rather than equality is to allow for an empty set of methods or fields. For readability and to make it easier for non-experts devising specifications, the name of class is used rather than the pointer into the class list which one might expect. Both are unique keys for the list.

Looking at the $Constructor predicate, a constructor is not available for primitive classes (here only int, if strings were added the constraint might be: $@class \in \{\text{“}\texttt{int}\text{”}\} \cup \{\text{“}\texttt{string}\text{”}\}$), so the first disjunct caters for this case. Otherwise the constructor pointer yields a cell containing the constructor code. It is assumed that a constructor is argument-free, which means that all constructors can fulfil the same specification and furthermore all classes can have a constructor because it merely allocates fresh objects for every field.

$$
\begin{aligned}
&\textsf{forall } \$\text{Pre}, \$\text{Post}.\\
&\textsf{recdef } \$\text{MethodLseg}(a, z; \%methods) := a = z \star \%methods = \emptyset\\
&\quad \vee \exists next, \%rest, @mName, arity, @targetClass, @argClass.\\
&\quad a \mapsto\\
&\quad \forall target, arg, \%cs, \%extraArgs, \%tFsPre.\\
&\qquad \{\ \$\text{Pre}(target, arg; \%tFsPre, \%cs, \%extraArgs; @targetClass, @mName, @argClass)\ \}\\
&\qquad\qquad \cdot(target, arg)\\
&\qquad \{\ \$\text{Post}(target, arg; \%tFsPre, \%cs, \%extraArgs; @targetClass, @mName, @argClass)\ \}\\
&\quad , @mName, @targetClass, @argClass, next\\
&\quad \star\ @mName \notin proj(2, \%rest)\\
&\quad \star\ \$\text{MethodLseg}(next, z; \%rest)\\
&\quad \star\ \%methods = \{(a, @mName, @targetClass, @argClass)\} \cup \%rest;
\end{aligned}
$$

Figure 3.6: Predicate describing method metadata on the heap

Therefore a constructor takes a single argument for storing a pointer to the freshly created object. The post-condition ensures that the object and its fields are allocated, by the instance of $References. The occurrence of $FieldsExist ensures that the object is well-defined with respect to the field declarations in the class definition (the class fields are passed with the parameter $\%cFs$)). The Hoare-triple contained in the $Constructor is deeply nested inside a class list, and it is parametrised by variables bound outside the triple.

3.3.2 Methods

An important concept for the methods is the way support for reflective invocation has been introduced through a *generic specification*. Obviously a program will consist of many methods, each with probably different behaviour, so how can they all be represented by the metadata? Firstly, we see the utility of nested triples again because the behaviour of a method is actually being included in the same predicate as the metadata. But the important question is what the pre- and post-condition should contain. In this approach, they are effectively parametrised by predicates that can be defined on a per example basis. That is to say that the reflective library is verified *for all* pre and post-conditions, to be given later for concrete cases. This is similar to Parkinson and Bierman's abstract predicates [9], where implementing classes give their own definition.

Methods have been restricted to having a single argument. This is important so that all methods can be described by a single nested triple (which does not support varying arities in this verification system). This does not hinder the programming possibilities because multiple arguments could be wrapped up into a single list which is later expanded in the method body. The metadata description of methods contains the name, the target

$$\mathsf{recdef}\ \$\mathrm{Pre}(\mathit{target}, \mathit{arg}; \%\mathit{tFsPre}, \%M, \%\mathit{extraArgs}; @\mathit{targetClass}, @\mathit{mName}, @\mathit{argClass}) :=$$
$$\exists \%\mathit{argFs}.\ \$\mathrm{Meta}(\%M) \star \$\mathrm{Object}(\mathit{target}; \%\mathit{tFsPre}; @\mathit{targetClass}) \star \$\mathrm{Object}(\mathit{arg}; \%\mathit{argFs}; @\mathit{argClass});$$

$$\mathsf{recdef}\ \$\mathrm{Post}(\mathit{target}, \mathit{arg}; \%\mathit{tFsPre}, \%M, \%\mathit{extraArgs}; @\mathit{targetClass}, @\mathit{mName}, @\mathit{argClass}) :=$$
$$\exists \%\mathit{argFs}, \%\mathit{tFsPost}.\ \$\mathrm{Meta}(\%M) \star \$\mathrm{Object}(\mathit{target}; \%\mathit{tFsPost}; @\mathit{targetClass}) \star \$\mathrm{Object}(\mathit{arg}; \%\mathit{argFs}; @\mathit{argClass}) \star \$\mathrm{Fun}(; \%\mathit{tFsPre}, \%\mathit{tFsPost});$$

$$\mathsf{recdef}\ \$\mathrm{Fun}(; \%\mathit{tFsPre}, \%\mathit{tFsPost}) := \quad // \text{ auxiliary predicate}$$
$$\exists n. \%\mathit{tFsPre} = \{(\mathit{target}+1, \texttt{''int''}, n)\} \star \%\mathit{tFsPost} = \{(\mathit{target}+1, \texttt{''int''}, n+1)\};$$

Figure 3.7: Example definitions for $Pre and $Post

class ($\simeq$ declaring class) and the argument's class. The list structure for method metadata is given in Figure 3.6, where the three pieces of metadata plus behavioural specification are stored for each element in the adjacent cells beginning at address a. The additional constraint on the list is that method names are unique. Recall that the metadata for each class keeps a separate method list, so it is still possible for two different classes to include methods of the same name.

At the point where the reflection library is utilized in a program the question arises about what the $Pre and $Post predicate definitions should look like. While the arguments of these predicates are fixed, it is possible to extend them by adding the extra arguments to the sixth argument ($\%\mathit{extraArgs}$). The parameters that are explicitly included are those which would seem to be most useful, or appear in the procedure's parameters or outside the nested procedure. The target and argument class parameters, along with method name, are bound outside of the triple in the method metadata. This ensures the method's types as described in the metadata are the same as the specification of the methods behaviour. In Figure 3.7, some minimal example definitions are given. The heap only needs to contain the target and an argument object, and after execution the post-condition shows that the target's fields have been updated. The precise manner in which the fields have changed is delegated to the predicate $Fun, where it is seen the value has incremented. In order to allow methods that are invoked reflectively to make further nested use of reflection, it is important that the $Pre and $Post definitions include the metadata predicate. This will become more clear in Section 3.4.3, where the specification and implementation of Method.invoke is shown, which can only be verified if this restriction is enforced.

The primary reason for the inclusion of the method name as an argument is so that the

$$\begin{array}{l}\mathsf{recdef}\ \$\mathrm{FieldLseg}(a, z; \%\mathit{fields}) := a = z \star \%\mathit{fields} = \emptyset \\ \vee\ \exists\ \mathit{next}, \%\mathit{rest}, @\mathit{fName}, @\mathit{class}, @\mathit{decClass}, \mathit{offset}. \\ a \mapsto @\mathit{fName}, @\mathit{class}, @\mathit{decClass}, \mathit{offset}, \mathit{next} \star @\mathit{fName} \notin \mathrm{proj}_2(\%\mathit{rest}) \\ \star\ \$\mathrm{FieldLseg}(\mathit{next}, z; \%\mathit{rest}) \\ \star\ \ \%\mathit{fields} = \{(a, @\mathit{name}, @\mathit{type}, @\mathit{decClass}, \mathit{offset})\} \cup \%\mathit{rest}\end{array}$$

Figure 3.8: Predicate definition for a list of fields contained within a class

$$\begin{array}{l}\mathsf{const}\ \mathit{META}; \\ \mathsf{recdef}\ \$\mathrm{Meta}(; \%cs) := \exists\, \mathit{list}.\ \mathit{META} \mapsto \mathit{list} \star \$\mathrm{ClassLseg}(\mathit{list}, 0; \%cs)\end{array}$$

Figure 3.9: Top-level predicate encapsulating the metadata

$Pre and $Post predicates can be indexed by individual methods (by using disjunctions, including the declaring class) which means that every method can have its own specification. However, this is not as much of a burden as it may seem. It is quite likely that only a small subset of the methods declared in a program are intended to be reflectively invoked. As such it is only necessary to engineer the $Pre and $Post definitions to cover those few methods. For any other method, they need not fulfil the generic specification. Secondly, in some example cases several methods can fit the same specification pattern, because the reflection might be used to invoke from a choice of methods which have similar functionality. For an example of this see the visitor pattern example in Chapter 4.

3.3.3 Fields

The final aspect of the metadata stores a description of class' fields. The list of fields contained in an element of the class metadata is defined by the predicate $FieldLseg in Figure 3.8. The description of fields contains the name (which must be unique), the type, a link to the declaring class containing this field, and an offset. This offset is an integer that can be added to the object's address to dereference the field's value. For instance, for this (unfolded) object at address o

$$o \mapsto @\mathit{clsA},\ \ 1, f_1,\ \ 1, f_2,\ \ 1, f_3,\ \ 0$$

the offset of the first field would be 2 (the field is accessed by $[o + 2]$), the second 4, and the third 6.

3.3.4 Metadata container

At the top-level, the metadata is all wrapped up into a single predicate, defined in Figure 3.9. Any procedure that wants to perform reflection must carry the instance of the

$Meta predicate in its specification, although there should be no need to ever unfold this predicate in the main program because the data contained within will be accessed through the reflective library.

Because the metadata is essentially a list of classes, the top-level predicate just needs to contain this list. A reserved constant *META* is declared, which is a pointer to the start of the class list, whose end is 0.

3.4 Reflective library

With a representation of the metadata on the heap now available, as presented in the previous section, it is possible to implement the reflective library procedures. The body of these procedures uses primitive heap manipulation operations, and standard techniques for linked list traversal. As such the program code is short, however there are a number of annotations for guiding the unfolding/refolding of predicates.

The following subsections describe each supported reflective procedure, grouped by metadata class. The code bodies are performing standard linked list manipulation, and so some have been omitted and others can be skipped during reading. The specifications are arguably more revealing.

3.4.1 Object

Object_getClass

proc $Object\_getClass(obj, res)$

$\forall\ clsPtr, \%clsFs, \%clsMs, construct, @clsName, \%objFs, \%cs, case.$

pre : $\$\text{Object}(obj; \%objFs; @clsName) \star res \mapsto \_ \star \$\text{Meta}(; \%cs)$

$\star\ (clsPtr, @clsName, \%clsFs, \%clsMs, construct) \in \%cs \star case = 1$

$\vee\ \$\text{Object}(obj; \%objFs; @clsName) \star res \mapsto \_ \star \$\text{Meta}(; \%cs)$

$\star\ @clsName \in \text{proj}_2(\%cs) \star case = 2;$

post :

$\$\text{Object}(obj; \%objFs; @clsName) \star res \mapsto clsPtr \star \$\text{Meta}(; \%cs)$

$\star\ (clsPtr, @clsName, \%clsFs, \%clsMs, construct) \in \%cs \star case = 1$

$\vee\ \exists\ clsPtr, clsIntFs, \%clsFs, \%clsMs, construct.$

$\$\text{Object}(obj; \%objFs; @clsName) \star res \mapsto clsPtr \star \$\text{Meta}(; \%cs)$

$\star\ (clsPtr, @clsName, \%clsFs, \%clsMs, construct) \in \%cs \star case = 2;$

{

locals $@id, classList$;

ghost "unfold $\$\text{Object}(obj; ?; ?)$";

// Each object is enriched with its type in the first cell

$@id := [obj]$;

ghost "fold $\$\text{Object}(obj; \%objFs; ?)$";

ghost "unfold $\$\text{Meta}(; \%cs)$";

$classList := [META]$;

call $searchClassList(classList, @id, res)$;

ghost "split $\$\text{ClassLseg}\ classList\ (?, ?, ?, ?, ?)$";

ghost "fold $\$\text{ClassLseg}(?, ?; ?)$";

ghost "join $\$\text{ClassLseg}\ classList$";

ghost "fold $\$\text{Meta}(; \%cs)$"

}

This procedure retrieves a pointer into the metadata referencing the record for the class of the given object. There are effectively two specifications, a stronger one and a weaker one, with the pre- and post-conditions connected by the *case* variable. In both cases, the specification states that the first argument is an object, the second argument is a pointer to a cell (at which the result will be stored), and the metadata is available. The specifications then differ in the last constraint, depending on how much is known about the class. In the

first case, the full tuple is available, but in the weaker second case it is only required that the class appears in the metadata. The post-condition for the second case then asserts that the full tuple is contained in the metadata, with some existential variables. The result of this procedure is the pointer to the class in the metadata class list, which is the first element of the tuple.

The body of the procedure unfolds the object predicate to expose the class contained within. Because this is actually the name of the class, some extra work is required to find the respective class in the metadata. The metadata is unfolded from the top-level $Meta predicate and the start of the class list retrieved by dereferencing the constant *META*. An auxiliary procedure is then used which uses trivial linked list traversal to locate an element using the class name as the key. The sequence of four ghost statement annotations at the end serve to expose the fact that class names are unique, which is hidden in the class predicate definition so that the "found" class record's tuple can be asserted to be identical to that in the stronger pre-condition.

Note that having the two specifications is redundant because the class name is unique in the class list anyway, so the "stronger" case can be inferred from the "weaker" by the constraints hidden inside the metadata predicates (as seen by the four ghost statements at the end). However, the purpose of the reflective library is to perform all the interfacing with the metadata such that the metadata need not be inspected from procedures outside of the library.

3.4.2 Class

Class_getName

proc $Class\_getName(clsPtr, res)$
$\forall \%cs, @clsName, \%clsFs, \%clsMs, construct.$
pre : \$Meta$(;\%cs) \star (clsPtr, @clsName, \%clsFs, \%clsMs, construct) \in \%cs$
$\star\ res \mapsto \_$;
post : \$Meta$(;\%cs) \star res \mapsto @clsName$;
{
ghost "unfold \$Meta(;?)";
ghost "split \$ClassLseg *list1* $(clsPtr, @clsName, \%clsFs, \%clsMs, construct)$";
// The name is in the first record of the class metadata
$[res] := [clsPtr]$;
ghost "fold \$ClassLseg$(clsPtr, 0; ?)$";
ghost "join \$ClassLseg *list1*";
ghost "fold \$Meta$(;\%cs)$"
}

The procedure for retrieving the string name of a class takes a pointer to an element in the class list as the first argument, and stores the corresponding name for that record in the result cell. Because the pointer to the relevant element is provided, the code can be a single line which dereferences at the name record which is the first cell (recall a class record looks like $a \mapsto @name, ms, fs, cons, next$). The ghost annotations expose the relevant position in the list.

Class_forName

```
proc Class_forName(@cName, res)
∀ cPtr, %cMs, construct, %cs, %objFs.
  pre : $Meta(; %cs) ⋆ (cPtr, @cName, %objFs, %cMs, construct) ∈ %cs
      ⋆ res ↦ _;
  post : $Meta(; %cs) ⋆ res ↦ cPtr;
{
  locals @id, classList;
  ghost "unfold $Meta(; ?)";
  classList := [meta];
  call searchClassList(classList, @cName, res);
  ghost "split $ClassLseg classList (?, ?, ?, ?, ?)";
  ghost "fold $ClassLseg(?, ?; ?)";
  ghost "join $ClassLseg classList";
  ghost "fold $Meta(; ?)"
}
```

Aside from Object_getClass, the other way to access the handle of a class is to look one up by class name. The pre-condition requires that there is a class of the relevant name defined in the metadata, and the postcondition asserts that the pointer has been stored in the result cell. The body of the procedure uses searchClassList to iterate through the class list, looking for the element whose name matches the search needle. As with Object_getClass, there is an unfolding/refolding at the end to expose the fact that the found class pointer must be equal to that in the pre-condition because the names are unique in the list.

Class_getMethod

proc $Class\_getMethod(clsPtr, @name, res)$

$\forall$ @clsName, %clsFs, %clsMs, construct, %cs.

pre : \$Meta(; %cs) $\star$ (clsPtr, @clsName, %clsFs, %clsMs, construct) $\in$ %cs

$\star$ res $\mapsto$ _;

post : $\exists$ a, @argCls.

\$Meta(; %cs) $\star$ (clsPtr, @clsName, %clsFs, %clsMs, construct) $\in$ %cs

$\star$ res $\mapsto$ a $\star$ (a, @name, @clsName, @argCls) $\in$ %clsMs

$\vee$ \$Meta(; %cs) $\star$ (clsPtr, @clsName, %clsFs, %clsMs, construct) $\in$ %cs

$\star$ res $\mapsto$ 0 $\star$ @name $\notin$ proj_2(%clsMs);

```
{
  locals methodList;
  ghost "unfold $Meta(; ?)";
  ghost "split $ClassLseg list1 (?, ?, ?, ?, ?)";
  // Method list is at the second record of our class metadata
  methodList := [clsPtr + 1];
  call searchMethodList(methodList, clsPtr, @name, res);
  ghost "fold $ClassLseg(clsPtr, 0; ?)";
  ghost "join $ClassLseg list1";
  ghost "fold $Meta(; ?)"

}
```

Given a pointer to a class in the metadata, which contains a list of methods, it is possible to search for a method by name. In this specification, the post-condition allows the procedure to fail in the event that the sought method does not exist (equivalent to Java's NoSuchMethodException). Note that this does not mean that both cases need to be handled when the procedure is used in a program where it is known that the method *does* exist, because an inconsistency would be introduced by the $\notin$ constraint. This allows the discounting of the case modelled in Java by the NoSuchMethodException when it is known that the method does exist. The body of the procedure uses searchMethodList, which performs a standard linked list search in the method metadata lists.

Class_getConstructor

proc $Class\_getConstructor(clsPtr, res)$

$\forall$ @*clsName*, %*clsMs*, %*cs*, %*objFs*, *construct*.

pre : \$Meta(; %*cs*) $\star$ (*clsPtr*, @*clsName*, %*objFs*, %*clsMs*, *construct*) $\in$ %*cs*

$\star$ *res* $\mapsto$ _;

post : \$Meta(; %*cs*) $\star$ *res* $\mapsto$ *construct*;

{

ghost "unfold \$Meta(; ?)";

ghost "split \$ClassLseg *list1* (?, ?, ?, ?, ?)";

[*res*] := [*clsPtr* + 3];

ghost "fold \$ClassLseg(?, ?; ?)";

ghost "join \$ClassLseg *list1*";

ghost "fold \$Meta(; ?)"

}

This procedure returns a pointer to the constructor code. Recalling that primitive classes do not have a reflective constructor, it can be noted that this specification is still suitable for the primitive classes because the value of *construct* in the class list tuple will be zero/nil. When unfolding the \$Constructor predicate, such a value will not yield a nested specification. This procedure's single-line body is almost identical to Class_getName, except that the constructor record is at the fourth cell.

Class_getDeclaredFields

recdef $FieldList$(a; \%fields) := a = 0 \star \%fields = \emptyset$

$\vee \exists fPtr, @fName, @fType, @fDecClass, offset, next, \%rest.$

$a \mapsto fPtr, next \star$ $FieldList$(next; \%rest)$

$\star\ \%fields = \{(fPtr, @fName, @fType, @fDecClass, offset)\} \cup \%rest$

$\star\ @fName \notin \text{proj}_2(\%rest);$

proc $Class\_getDeclaredFields(cPtr, res)$

$\forall construct, @cName, \%cMs, \%cs, \%objFs.$

pre : $Meta$(; $\%cs$) $\star$ $(cPtr, @cName, \%objFs, \%cMs, construct) \in \%cs$

$\star\ res \mapsto \_;$

post : $\exists fs.$$Meta$(; $\%cs$) $\star$ $FieldList$(fs; $\%objFs$) $\star\ res \mapsto fs;$

{

locals $fields$;

ghost "unfold $Meta(;?)$";

ghost "split $ClassLseg $list1(?,?,?,?,?)$";

$fields := [cPtr + 2]$;

call $copyFieldLseg(fields, res)$;

ghost "fold $ClassLseg$(?,?;?)";

ghost "join $ClassLseg $list1$";

ghost "fold $Meta(;?)$"

}

proc $copyFieldLseg(l, res)$

$\forall \%fs.$

pre : $FieldLseg$(l, 0; $\%fs$) $\star\ res \mapsto_;$

post : $\exists cp.$$FieldLseg$($l$, 0; $\%fs$) $\star$ $FieldList$(cp; $\%fs$) $\star\ res \mapsto cp;$

This procedure returns a list of the fields in the metadata for the given class. Rather than simply providing a reference to the field list contained within the metadata, the result list is a new list (a copy). This is in common with the aim that metadata is only accessed from within the library. Furthermore, creating a copy of the list here allows the returned list of fields to be changed (for example as an iterator, removing seen elements), whilst ensuring that the metadata remains unaltered. For instance, one might like to iterate through the list and remove seen elements after each iteration, or filter the list by removing fields that do not match some condition. The resulting list is a list of pointers into the metadata, that is it is a copy of the *fPtr* part of the $FieldLseg structure only.

There is an auxiliary procedure for creating a copy of a metadata $FieldLseg to a $FieldList (and a dispose procedure for use in the utilizing programs).

Class_getDeclaredField

proc $Class\_getDeclaredField(cPtr, @fName, res)$

$\forall\, construct, @cName, \%cMs, \%cs, \%objFs.$

pre : $\$\text{Meta}(;\%cs) \star (cPtr, @cName, \%objFs, \%cMs, construct) \in \%cs$

$\star\ @fName \in proj(2, \%objFs) \star res \mapsto \_;$

post : $\exists\, fPtr, @fType, offset.$

$\$\text{Meta}(;\%cs) \star (fPtr, @fName, @fType, @cName, offset) \in \%objFs$

$\star\ res \mapsto fPtr;$

{

locals *fields*;

ghost "unfold $\$\text{Meta}(;\%cs)$";

ghost "split $\$\text{ClassLseg}\ list1(cPtr, @cName, \%objFs, \%cMs, construct)$";

$fields := [cPtr + 2];$

call $searchFieldList(fields, @fName, res);$

ghost "fold $\$\text{ClassLseg}(cPtr, ?; ?)$";

ghost "join $\$\text{ClassLseg}\ list1$";

ghost "fold $\$\text{Meta}(;\%cs)$";

}

If both the class and name of a field declared within that class are known, the pointer to that field in the metadata can be obtained with this procedure. In the same way that Class_getMethod works, this uses an auxiliary list search procedure to iterate through the field list contained in the class attempting to find a matching field.

Class_isPrimitive

proc $Class\_isPrimitive(cPtr, res)$

$\forall\, \%cs, @cName, \%cFs, \%cMs, construct.$

pre : $\$\text{Meta}(;\%cs) \star (cPtr, @cName, \%cFs, \%cMs, construct) \in \%cs \star res \mapsto \_;$

post : $\$\text{Meta}(;\%cs) \star res \mapsto 0 \star @cName \neq$ "`int`"

$\vee\ \$\text{Meta}(;\%cs) \star res \mapsto 1 \star @cName =$ "`int`";

This procedure returns a result of 1 or 0, representing true or false, depending on whether the given class is considered primitive. Presently there is only the primitive integer class (named "int"), which is used in the post-condition to assert the result is as expected.

3.4.3 Method

Method_invoke

```
proc Method_invoke(method, target, arg)
∀ @targetCls, @argCls, %cs, %otherArgs, @mName,
   tClsPtr, %tClsFs, %tClsMs, tCons.
  pre :
   (method, @mName, @targetCls, @argCls) ∈ %tClsMs ⋆
   (tClsPtr, @targetCls, %tClsFs, %tClsMs, tCons) ∈ %cs ⋆
   $Pre(target, arg; %tFsPre, %cs, %otherArgs; @targetCls, @mName, @argCls);
  post :
   $Post(target, arg; %tFsPre, %cs, %otherArgs; @targetCls, @mName, @argCls);
{
 // Note that invoke is really just an EVAL.
 eval [method](target, arg)
  "before lookup // Hint to the verifier to 'expose' nested triple in MethodLseg
   unfold $Pre(target, arg; %tFsPre, %cs, %otherArgs; @targetCls, @mName, @argCls),
   unfold $Meta(; %cs),
   split $ClassLseg list1 (tClsPtr, @targetCls, %tClsFs, %tClsMs, tCons),
   split $MethodLseg ms2 (method, @mName, @targetCls, @argCls)"
}
```

Whilst reasoning about method invocation may be complex because of the underlying behaviour, the invoke implementation is simple when using this metadata representation. In fact, given that the behaviour is described by a nested triple contained within the method metadata, it is merely an eval command. As discussed in Section 3.3, the metadata must be contained within the concrete definitions of the $Pre and $Post predicates. The specification is the same as that contained within the method metadata, with the addition of the two constraints in the pre-condition that distinguish the particular method in the particular class.

The annotations preceded by before lookup guide the prover to perform the desired sequence of predicate unfoldings in order to locate the behavioural specification for the code stored at address *method*. Without the hints, the specification may still be discovered by the prover's built-in "unintelligent" choice of predicates to unfold, however in a much

slower fashion. Additionally, they are included for presentation to highlight the structure of the underlying metadata.

3.4.4 Constructor

Constructor_newInstance

$$
\begin{array}{l}
\mathsf{proc}\ Constructor\_newInstance(this, res) \\
\forall\ clsPtr, @clsName, \%clsObjFs, \%clsMs, \%cs. \\
\quad \mathsf{pre} : \$\mathrm{Meta}(;\%cs) \star (clsPtr, @clsName, \%clsObjFs, \%clsMs, this) \in \%cs \\
\qquad \star\ @clsName \neq \text{“}\texttt{int}\text{”} \star res \mapsto \_; \\
\quad \mathsf{post} : \exists\, o, \%fieldRefs, \%objFieldRefs, \%objFs. \\
\qquad \$\mathrm{Meta}(;\%cs) \star res \mapsto o \star \$\mathrm{Object}(o; \%objFs; @clsName) \\
\qquad \star\ \$\mathrm{References}(;\{(o, @clsName, \%objFs)\}, \%objFieldRefs) \\
\qquad \star\ \$\mathrm{FieldRefs}(;\%objFs, \%fieldRefs) \\
\qquad \star\ \$\mathrm{FieldsExist}(o; \%clsObjFs, \%objFs) \star \%fieldRefs \subseteq \%objFieldRefs; \\
\{ \\
\quad \mathsf{eval}[this](res) \\
\}
\end{array}
$$

The function of a reflectively invokable constructor is to create a new object, and initialize every respective field with fresh objects. Recall that this interfaces with an argument-free constructor only. Although the concrete constructor declared in the program may be stronger, for example containing further constraints on the precise initial value of fields, when invoking the constructor reflectively the reasoning can only use the weaker behaviour so as to support a greater number of concrete constructors. As with Method_invoke, the implementation is essentially a wrapper for an eval, and the specification is as in the $Constructor predicate.

3.4.5 Field

Field_getType

proc $Field\_getType(f, res)$

$\forall\ cPtr, fOffset, @class, @fName, @fClass, \%cFs, \%cMs, \%cs, construct,$
$fCPtr, \%fCFs, \%fCMs, fCon.$

pre : $\$\mathrm{Meta}(;\%cs) \star (cPtr, @class, \%cFs, \%cMs, construct) \in \%cs$
$\star\ (f, @fName, @fClass, @class, fOffset) \in \%cFs$
$\star\ (fCPtr, @fClass, \%fCFs, \%fCMs, fCon) \in \%cs \star res \mapsto \_;$

post : $\$\mathrm{Meta}(;\%cs) \star res \mapsto fCPtr;$

{

locals $classes, @type;$

ghost "unfold $\$\mathrm{Meta}(;\%cs)$";

ghost "split $\$\mathrm{ClassLseg}list1(?,?,?,?,?)$";

ghost "split $\$\mathrm{FieldLseg}fs3(?,?,?,?,?)$";

$@type := [f + 1];$

ghost "fold $\$\mathrm{FieldLseg}(?,?;?)$";

ghost "join $\$\mathrm{FieldLseg}fs3$";

ghost "fold $\$\mathrm{ClassLseg}(cPtr, 0;?)$";

ghost "join $\$\mathrm{ClassLseg}list1$";

$classes := [meta];$

call $searchClassList(classes, @type, res);$

}

When a pointer into the field list of a certain class has been obtained, the class type record of the field can be accessed by this procedure. The pre-condition requires that the field's declaring class is in the metadata, along with the field given as an argument being contained within that class's field list. Additionally, the class of the field (which is the result) must also be contained in the metadata. Due to the encoding used for the reflective reasoning using the string name of classes to refer to the class, some extra work needs to be done to retrieve the pointer to the class in the metadata, based on that name. That is why the searchClassList procedure, first seen with Object_getClass, is needed again.

Field_getDeclaringClass

$$
\begin{array}{l}
\mathsf{proc}\ \mathit{Field\_getDeclaringClass}(f, \mathit{res}) \\
\quad \forall\ \mathit{cPtr}, \mathit{fOffset}, @\mathit{fName}, @\mathit{fType}, @\mathit{decClass}, \%\mathit{cFs}, \%\mathit{cMs}, \%\mathit{cs}, \mathit{construct}. \\
\quad \mathsf{pre} : \$\mathrm{Meta}(;\%\mathit{cs}) \star (\mathit{cPtr}, @\mathit{decClass}, \%\mathit{cFs}, \%\mathit{cMs}, \mathit{construct}) \in \%\mathit{cs} \\
\qquad \star\ (f, @\mathit{fName}, @\mathit{fType}, @\mathit{decClass}, \mathit{fOffset}) \in \%\mathit{cFs} \star \mathit{res} \mapsto \_; \\
\quad \mathsf{post} : \$\mathrm{Meta}(;\%\mathit{cs}) \star \mathit{res} \mapsto \mathit{cPtr};
\end{array}
$$

In a similar way to the previous method, the record containing the class which declared the given method can be retrieved. This again must be converted from the name of the class to a pointer in the metadata.

Field_getName

$$
\begin{array}{l}
\mathsf{proc}\ \mathit{Field\_getName}(f, \mathit{res}) \\
\quad \forall\ \mathit{cPtr}, \mathit{fOffset}, @\mathit{cName}, @\mathit{fName}, @\mathit{fType}, \%\mathit{cFs}, \%\mathit{cMs}, \%\mathit{cs}, \mathit{construct}, \mathit{intFs}. \\
\quad \mathsf{pre} : \$\mathrm{Meta}(;\%\mathit{cs}) \star (\mathit{cPtr}, @\mathit{cName}, \%\mathit{cFs}, \%\mathit{cMs}, \mathit{construct}) \in \%\mathit{cs} \\
\qquad \star\ (f, @\mathit{fName}, @\mathit{fType}, @\mathit{cName}, \mathit{fOffset}) \in \%\mathit{cFs} \star \mathit{res} \mapsto \_; \\
\quad \mathsf{post} : \$\mathrm{Meta}(;\%\mathit{cs}) \star \mathit{res} \mapsto @\mathit{fName};
\end{array}
$$

The simplest procedure for reflecting on fields yields the name of the given field. There is no extra work here so the code is a single line, and the annotations unfold and refold the metadata at the relevant point.

Field_get

proc $Field\_get(f, instance, res)$

$\forall\, cPtr, @cName, @fName, @fType, fOffset, \%cFs, \%cMs, \%cs, \%instanceFs,$
$construct, \%otherFs, value.$

pre : $\$\mathrm{Meta}(;\%cs) \star (cPtr, @cName, \%cFs, \%cMs, construct) \in \%cs$
$\star\ (f, @fName, @fType, @cName, fOffset) \in \%cFs$
$\star\ \$\mathrm{Object}(instance; \%instanceFs; @cName)$
$\star\ (instance + fOffset, @fType, value) \in \%instanceFs \star res \mapsto \_;$

post :
$\$\mathrm{Meta}(;\%cs) \star \$\mathrm{Object}(instance; \%instanceFs; @cName) \star res \mapsto value;$

{

locals $offset, oFields$;

$offset := [f + 3]$;

$oFields := instance + 1$;

call $searchObjFields(offset, oFields, res)$;

}

proc $searchObjFields(offset, list, res)$

$\forall\, \%fs, @fType, value.$

pre : $\$\mathrm{ObjFields}(list; \%fs) \star (list - 1 + offset, @fType, value) \in \%fs \star res \mapsto \_;$

post : $\$\mathrm{ObjFields}(list; \%fs) \star res \mapsto value;$

{

locals $decr, next$;

$decr := offset - 1$;

ghost "unfold $\$\mathrm{ObjFields}(list; \%fs)$";

if $decr = 0$ then{

$[res] := [list]$

}else{

$next := list + 1$;

call $searchObjFields(decr, next, res)$

};

ghost "fold $\$\mathrm{ObjFields}(list; \%fs)$"

}

The current value of a given field in an object can be accessed by looking up the offset of the field in the metadata, and then using that offset to dereference the field contents

location. Unfortunately, while the address of the field is known (object pointer plus the offset), a recursive auxiliary procedure is used because the $ObjFields predicates need to be unfolded a varying number of times (depending on the offset). Without first unfolding, the dereferencing cannot take place because the cell is hidden in the symbolic state. An alternative approach would be to use a (recursive) lemma to "split" the list represented by the $ObjFields predicate. Whilst this would not reduce the workload on the human verifier, as the lemma would need writing and proving, the approach would have the advantage that the implementation is not changed to support the specification.

The specification includes the constraint that the field exists in the object, and with its type and offset being the same as in the metadata. The ghost statements are omitted.

Field_set

proc $Field\_set(f, instance, newValue)$

$\forall\, cPtr, @cName, @fName, @fType, fOffset, \%cFs, \%cMs, \%cs, \%instanceFs,$
$construct, \%otherFs, value.$

pre : $\$\text{Meta}(;\%cs) \star (cPtr, @cName, \%cFs, \%cMs, construct) \in \%cs$
$\star\ (f, @fName, @fType, @cName, fOffset) \in \%cFs$
$\star\ \$\text{Object}(instance; \%instanceFs; @cName)$
$\star\ \%instanceFs = \{(instance + fOffset, @fType, value)\} \cup \%otherFs$
$\star\ \%otherFs = \%instanceFs \setminus \{(instance + fOffset, @fType, value)\};$

post : $\exists\, \%newFs.$
$\$\text{Meta}(;\%cs) \star \$\text{Object}(instance; \%newFs; @cName)$
$\star\ \%newFs = \{(instance + fOffset, @fType, newValue)\} \cup \%otherFs;$

{
locals $offset, oFields;$
$offset := [f + 3];$
$oFields := instance + 1;$
call $updateObjField(oFields, offset, newValue)$
}

proc $updateObjField(fields, offset, newValue)$
$\forall\, @type, old\_value, \%fs, \%otherFs.$
pre : $\$\text{ObjFields}(fields; \%fs) \star (fields - 1 + offset, @type, old\_value) \in \%fs$
$\star\ \%otherFs = \%fs \setminus \{(fields - 1 + offset, @type, old\_value)\};$
post : $\$\text{ObjFields}(fields; \{(fields - 1 + offset, @type, newValue)\} \cup \%otherFs);$

Almost identical to the previous procedure for *reading* the value of a field is the procedure for updating the value. Again, the offset is used and the fields "traversed" recursively, with the only difference being that the auxiliary procedure instead stores the *newValue*. The specification is more different, which shows that the pertinent field is updated, but all the other fields (%*otherFs*) remain unchanged.

3.5 Automated generation of metadata

3.5.1 Introduction

At this point, it has been shown how one can represent the metadata on the heap and use a set of reflective library procedures to access it. However, there is one more stage to supporting reflection: setting up the metadata for a particular program, or populating the lists on the heap. An earlier phase of this work required that the metadata be manually created *within the main program code* [58]. This was achieved by constructing the linked lists, with the features for higher-order store used to store the methods with the metadata. Naturally this is tedious, and the created metadata might not correctly represent the actual program as the metadata could be "faked".

A simple demonstrative example of the steps needed to manually create the metadata can be seen in Figure 3.10. Here a small class "A" is shown (making use of another empty class "B", the metadata setup of which is omitted), which includes two fields and one method. Note that the constructors' post-conditions use concrete set constructions to show the fields created are those needed by the class definition. They do not include the metadata in their specifications, but this works because the check that the first argument of $FieldsExist matches the metadata will take place when folding the $ClassLseg predicate.

The metadata is created in the body of *main*, which begins by creating the method list. Recall that each method record includes the procedure and its name. The cells starting at *msA* are then folded into a method list predicate. Next the fields are declared and folded into the predicate. Thirdly, the constructor is added, which is checked that it meets the generic constructor specification (in $Constructor, Figure 3.5) during the fold operation. Finally the class record is created, which includes the three parts that were allocated above, plus the name. The fold of the class list predicate will then check that the class types used by the methods and constructor match with the class.

The code for setting up the metadata is tedious, and the pattern is repeated for every class in the program, which may be many, meaning quite an undertaking is necessary before the program can be verified. Furthermore this has required additional code be inserted in the main procedure, which destroys the separation of program code from reasoning because it is more than just annotations to the source code. A final disadvantage is that the actual names of the classes and methods are not checked to be those used in the metadata, although this is safe in the sense that all class types will be consistent in the specifications and metadata.

```
// Assumes two classes used in the program:
// class A { int x; B y; foo(arg) { ... }}
// class B { }
proc main()
  pre : META ↦ 0;
  post : ∃%cs. $Meta(;%cs);
{
  locals msA, fsA_y, fsA, consA, clsA, ...;
  msA := new 0,"foo",0;
  [msA] := A_foo(_, _);
  ghost 'fold $MethodLseg(0, 0; ∅)';
  ghost 'fold $MethodLseg(msA, 0; {(msA,"foo","A","B")})';

  fsA_y := new "y","B","A",3,0;
  ghost 'fold $FieldLseg(0, 0; ∅)';
  ghost 'fold $FieldLseg(fsA_y, 0; {(fsA_y,"y","B","A",3)})';
  fsA := "x","int","A",1,fsA_y;
  ghost 'fold $FieldLseg(fsA, 0; {(fsA,"x","int","A",1)}
                ∪{(fsA_y,"y","B","A",3)})';
  consA := new 0;
  [consA] := A_construct(_);
  ghost 'fold $Constructor(consA; ????;"A")';
  clsA := new "A", msA, fsA, consA, 0;
  ghost 'fold $ClassLseg(0, 0; ∅)';
  ghost 'fold $ClassLseg(clsA, 0; {(clsA,"A", fsA, msA, consA)})';

  ... // ditto for class B
  // Start of program code:
  ...
}
```

Figure 3.10: An abbreviated example showing manual creation of metadata in the program code

A better solution is to have the metadata automatically generated such that the list argument of the $Meta predicate is populated based on recognized patterns in the parsed source program. The linked lists' construction can be assumed to have taken place in the background.

To trigger the generation of metadata, the tool looks for the existance of the formula $META \mapsto 0$ in the pre-condition of the *main* procedure, where *META* was a constant declared in the library in Section 3.3. Additionally, the reflective library must be "loaded", which is achieved simply with a configuration flag to the tool which has the effect of adding all the library procedure and predicate declarations to the program being verified.

The semantics here follows the Java VM design, whereby the metadata is created when a class is loaded and garbage collected when a class is unloaded. Because there is no concept of class loading here, *all* the classes are effectively thought to be "loaded" right before the program is executed, so the metadata is stored on the heap before the symbolic execution of *main*.

The importance of adding strings to the language will now become clear, because the metadata should be a true representation of the procedures contained within the program, such that they can be looked up by strings. Because this lookup may be by concatenation, it would be non-trivial to implement an integer encoding.

3.5.2 Recognition of classes

The predicates defined in Section 3.3 give the data structures for representing the metadata. The goal here is then to take the input program, and create an instance of the $Meta predicate. It is necessary to settle on the syntax conventions for how a program matches the class/method model. This is important because the class/object system is only rudimentarily modelled here, so there are no explicit class declarations.

Classes are determined by examining the names of all procedures declared in the program and matching them with the pattern Class_method. The name of the class is the string beginning with an uppercase letter preceding the underscore.

Methods are recognised at the same time as classes above, taking the name from the string following the underscore. The entailment is then checked between the procedure and the generic specification which appears in $MethodLseg. If the entailment succeeds, then method is added to the metadata. Otherwise it is ignored. The argument types are available through the final two arguments of the $Pre and $Post predicates.

Constructors are discovered as procedures which match the naming convention Class_construct (where the underline signifies exact name match, rather than a placeholder variable). In a similar approach to method handling, the specification of a constructor candidate is checked that it fits the generic specification given in the $Construct predicate definition.

$$\textsc{MainWithMeta}\quad \frac{\Pi;\Gamma \Vdash \{\Psi \star \$\mathrm{Meta}(\alpha)\}\, \mathcal{F}_{main}(params(\mathcal{F}_{main}))\, \{Q\}}{\Pi;\Gamma \Vdash \{\Psi \star META \mapsto 0\}\, \mathcal{F}_{main}(params(\mathcal{F}_{main}))\, \{Q\}} \quad \alpha = generateMeta(\Pi, \Gamma, \emptyset)$$

Figure 3.11: New rule for verifying the main procedure with pre-condition extended by metadata

Fields can be discovered from constructor's specifications. The instance of a $FieldsExist predicate contains in its first argument the required names and types of fields for the respective class. Note that if there is no constructor declared, then there can be no reflection on fields.

Once the program has been parsed and the metadata generated, the pre-condition of *main* is altered by replacing $META \mapsto 0$ with $\$\mathrm{Meta}(;\ldots)$ for some concrete set argument representing the list of classes. This is reflected by the new rule in Figure 3.11. The function $generateMeta(\Pi, \Gamma, \emptyset)$ examines the syntax of all procedures in the procedure context Γ, as described above, and builds a set of the form expected by $Meta. The function is defined as follows:

$$generateMeta(\Pi,\ (),\ \alpha) = \alpha$$

$$generateMeta(\Pi,\ \big(\Gamma, (\mathsf{proc}\ \mathcal{F}(\vec{x})\ \forall \vec{v}.\ \mathsf{pre} : P;\ \mathsf{post} : Q; \{\ldots\})\big),\ \alpha) =$$
$$generateMeta(\Pi, \big(\Gamma, \big(\mathsf{proc}\ \mathcal{F}(\vec{x})\ \forall \vec{v}.\ \mathsf{pre} : P;\ \mathsf{post} : Q; \{\ldots\})\big),$$
$$\alpha \cup \{(a, \text{“X”}, \emptyset, \emptyset, cons)\})$$
where $a, cons$ fresh
if F matches X_f and “X” $\notin \pi_2(\alpha)$

$$generateMeta(\Pi,\ \big(\Gamma, \big(\mathsf{proc}\ \mathcal{F}(\vec{x})\ \forall \vec{v}.\ \mathsf{pre} : P;\ \mathsf{post} : Q; \{\ldots\})\big),\ \alpha) =$$
$$generateMeta(\Pi,\ \Gamma,\ (\alpha \setminus \{(a, \text{“X”}, \emptyset, \%ms, cons)\}) \cup \{(a, \text{“X”}, \%fs, \%ms, cons)\})$$
if F matches $X\_\underline{construct}$ and $\exists a, \%ms, cons.(a, \text{“X”}, \emptyset, \%ms, cons) \in \alpha$
and Q matches $\ldots \star \$\text{FieldsExist}(\_; \%fs, \_) \star \ldots$
and $\Pi(\$\text{Construct}) \Leftrightarrow \_ \vee e \mapsto B \star \_$ and $\forall \vec{v}.\ \{P\} \cdot (\vec{x})\ \{Q\} \Rightarrow B$

$$generateMeta(\Pi,\ \big(\Gamma, (\mathsf{proc}\ \mathcal{F}(\vec{x})\ \forall \vec{v}.\ \mathsf{pre} : P;\ \mathsf{post} : Q; \{\ldots\}\big),\ \alpha) =$$
$$generateMeta(\Pi, \Gamma, (\alpha \setminus \{(a, \text{“X”}, \%fs, \%ms, cons)\}) \cup \{(a, \text{“X”}, \%fs, \%ms', cons)\})$$
if F matches X_f and $\exists a, \%fs, \%ms, cons.(a, \text{“X”}, \%fs, \%ms, cons) \in \alpha$
and $P = \$\text{Pre}(\_, \_; \_, \_, \_; @targClass, @argClass)$
and $\%ms' = \%ms \cup \{(a', \text{“f”}, @targClass, @argClass)\}$ where a' fresh
and $\Pi(\$\text{MethodLseg}) \Leftrightarrow \ldots \vee e \mapsto B \star \ldots$ and $\forall \vec{v}.\ \{P\} \cdot (\vec{x})\ \{Q\} \Rightarrow B$
otherwise $generateMeta(\Pi,\ \Gamma,\ \alpha)$

Note that the metadata generation requires the predicate context in addition to the procedure context. This is because the definitions of the metadata predicates are required such that the method and constructor specifications can be checked.

3.6 Soundness

As a result of the choice made with respect to the representation of metadata on the heap and the implementation of the reflective procedures, much of the soundness comes for free, based on the core rules for heap manipulating programs which have already been proved sound. *The reflection library is verified automatically.* Soundness arguments do still need to be made relating to the metadata generation.

In order to prove the soundness of the metadata generation rule, MainWithMeta, the operational semantics must first be extended by a special handling for a call to the *main* procedure, the point at which the heap is extended with the metadata. Recall from the

introduction in Chapter 1, and with the full details in [52], that the operational semantics relates configurations of the form (C, s, h), for some code statement C, environment stack s, and heap h. There is a set of configurations Safe_n^γ, which are those configurations that do not reduce to the special aborting configuration abort (denoting runtime errors including memory faults) in n steps or less.

$$\begin{array}{rcl}
(\mathsf{call}\ \mathcal{F}(e_1,\ldots,e_n),\ s\cdot\eta,\ h) & \leadsto^\gamma & (C; \mathsf{return},\ s\cdot\eta\cdot\eta[x_1\mapsto[\![e_1]\!]_\eta,\ldots,x_n\mapsto[\![e_n]\!]_\eta, \\
& & \qquad y_1\mapsto 0,\ldots,y_m\mapsto 0],\ h) \\
& & \text{if } \mathcal{F}\in\mathsf{dom}(\gamma) \text{ and} \\
& & \quad \gamma(\mathcal{F}) = \mathsf{proc}\,(x_1,\ldots x_n)\{\mathsf{locals}\,y_1,\ldots,y_m; C\} \\
& & \text{and } (\mathcal{F}\neq\mathcal{F}_{main} \\
& & \quad \text{or } (\mathit{META}\in\mathsf{dom}(h) \text{ and } h(\mathit{META})\neq 0)) \\
(\mathsf{call}\ \mathcal{F}_{main}(\vec{e}), s\cdot\eta, h) & \leadsto & (\mathsf{call}\ \mathcal{F}_{main}(\vec{e}), s\cdot\eta, h'\cdot h^\gamma_{meta}) \\
& & \text{if } h = h'\cdot[\mathit{META}\mapsto 0]
\end{array} \tag{3.1}$$

The first case is the original semantics for call, with an additional condition that it does not apply for the main procedure unless the metadata has been initialised. The special heaplet denoted h^γ_{meta} and representing the metadata encapsulated in the \$Meta predicate is defined as follows:

Definition 13. For predicate environment $\pi \models \Pi$ and predicate environment $\gamma \models_\pi \Gamma$:

$$\forall\, n,\eta,\sigma.\ (n,\eta,\sigma,h^\gamma_{meta}) \models^\gamma_\pi \$\mathrm{Meta}(\alpha)$$

where $\alpha = \mathit{generateMeta}(\Pi,\Gamma,\emptyset)$.

The existence of h^γ_{meta} can be shown by structural induction on the procedure context Γ with the definition of *generateMeta*. The soundness of the rule for the metadata generation rule, MainWithMeta, follows trivially from the above operational semantics and definition, as is shown on the following page.

Theorem 14 (Soundness of metadata generation)**.** *The semantics of the rule require showing that:* $\forall\pi \models \Pi.\ \forall\gamma \models_\pi \Gamma.$
If $\models^\gamma_\pi (\Psi \star \$\mathrm{Meta}(\alpha),\ \mathsf{call}\ \mathcal{F}_{main}(params(\mathcal{F}_{main})),\ Q)$
then $\models^\gamma_\pi (\Psi \star META \mapsto 0,\ \mathsf{call}\ \mathcal{F}_{main}(params(\mathcal{F}_{main})),\ Q)$

Proof. The premise is equivalent to: for all η, σ, w, k:

$$w, \eta, \sigma \models^\gamma_k (\llbracket \Psi \star \$\mathrm{Meta}(\alpha) \rrbracket_\pi,\ \mathsf{call}\ \mathcal{F}_{main}(params(\mathcal{F}_{main})),\ \llbracket Q \rrbracket_\pi)$$

By this premise and Definition 6, it can be assumed:
For all $r \in UPred(H), m < n, h, s$,
if $(m, \eta, \sigma, h) \in \llbracket \Psi \star \$\mathrm{Meta}(\alpha) \rrbracket_\pi (w) \star i^{-1}(w)(\mathsf{emp}) \star r$ then:

(a') $(\mathsf{call}\ \mathcal{F}_{main}(params(\mathcal{F}_{main})), s \cdot \eta, h) \in \mathrm{Safe}^\gamma_m$

(b') For all $k \leq m, h', \eta'$, if $(\mathsf{call}\ \mathcal{F}_{main}(params(\mathcal{F}_{main})), s \cdot \eta, h) \rightsquigarrow^\gamma_k (\mathsf{skip}, s \cdot \eta', h')$ then $(m - k, \eta', \sigma, h') \in \bigcup_{w'} \llbracket Q \rrbracket_\pi (w \circ w') \star i^{-1}(w \circ w')(\mathsf{emp}) \star r$

To show the conclusion, it must be proved that for all $r, m < n, h, s$, if:

$$(m, \eta, \sigma, h) \in \llbracket \Psi \star META \mapsto \_ \rrbracket_\pi (w) \star i^{-1}(w)(\mathsf{emp}) \star r \tag{3.2}$$

then

(a) $(\mathsf{call}\ \mathcal{F}_{main}(params(\mathcal{F}_{main})), s \cdot \eta, h) \in \mathrm{Safe}^\gamma_m$

(b) For all $k \leq m, h', \eta'$, if $(\mathsf{call}\ \mathcal{F}_{main}(params(\mathcal{F}_{main})), s \cdot \eta, h) \rightsquigarrow^\gamma_k (\mathsf{skip}, s \cdot \eta', h')$ then $(m - k, \eta', \sigma, h') \in \bigcup_{w'} \llbracket Q \rrbracket_\pi (w \circ w') \star i^{-1}(w \circ w')(\mathsf{emp}) \star r$

By the interpretation of $\star$ it must be that $h = h_1 \cdot h_2 \cdot h_3 \cdot h_4$ where

(i) $(m, \eta, \sigma, h_1) \in \llbracket \Psi \rrbracket_\pi (w)$

(ii) $(m, \eta, \sigma, h_2) \in \llbracket META \mapsto 0 \rrbracket_\pi (w)$

(iii) $(m, \eta, \sigma, h_3) \in i^{-1}(w)(\mathsf{emp})$

(iv) $(m, \eta, \sigma, h_4) \in r$

By the new case in the operational semantics (3.1) and the above, to show (a) suffices to show:

$$(\mathsf{call}\ \mathcal{F}_{main}(params(\mathcal{F}_{main})), s \cdot \eta, h_1 \cdot h^\gamma_{meta} \cdot h_3 \cdot h_4) \in \mathrm{Safe}^\gamma_{m-1} \tag{3.3}$$

and similarly for (b):

$$\begin{array}{l} \text{For all } k \leq m - 1, h', \eta' \\ \text{if } (\mathsf{call}\ \mathcal{F}_{main}(params(\mathcal{F}_{main})), s \cdot \eta, h_1 \cdot h^\gamma_{meta} \cdot h_3 \cdot h_4) \rightsquigarrow^\gamma_k (\mathsf{skip}, s \cdot \eta', h') \\ \text{then } (m - 1 - k, \eta', \sigma, h') \in \bigcup_{w'} \llbracket Q \rrbracket_\pi (w \circ w') \star i^{-1}(w \circ w')(\mathsf{emp}) \star r \end{array} \tag{3.4}$$

By the definition of h^γ_{meta} in Definition 13 and interpretation of $\star$, it follows from (3.2) that:

$$(m - 1, \eta, \sigma, h_1 \cdot h^\gamma_{meta} \cdot h_3 \cdot h_4) \in \llbracket \Psi \star \$\mathrm{Meta}(\alpha) \rrbracket_\pi (w) \star i^{-1}(w)(\mathsf{emp}) \star r$$

This leads to a version of a (a') and (b') where h and m are substituted by $h_1 \cdot h^\gamma_{meta} \cdot h_3 \cdot h_4$ and $m - 1$ respectively, which is exactly the required (3.3) and (3.4).

□

3.7 Discussion

This chapter has presented a technique for supporting verification of reflective programs. This has been achieved by storing the metadata on the heap, and implementing the reflective library procedures using primitive heap manipulation operations. When a program uses reflection, the annotations need to include $META \mapsto 0$ in the main procedure, and definitions for the $Pre and $Post predicates should be provided if methods will be invoked reflectively. *These two items are the only additional considerations required to make use of the reflection reasoning*, which keeps the extra reflection aspects hidden from the human verifier who need not be burdened with gaining an understanding of the underlying implementation. Also, thanks to the modularity of separation logic, the entire library (procedures and predicates) can become abstract once it has been verified once, until it is modified, to save verification time.

In addition to the advantage of reducing the need for expert knowledge, the separation of the metadata by way of the library interface means that it would be possible to disallow the unfolding of the metadata (outside the reflection module) with a check during the parsing stage. This would mean that the metadata cannot be altered outside of the reflective library and so will always be representative of the given program structure it was originally generated for.

An earlier published incarnation of this work [58] included a list of parameter types for methods, rather than the version here where they are fixed at two. Operationally it is the same because the list was, in fact, always fixed at length two – assuming that multiple arguments must be wrapped up into a list – however the list was used to allow for further future flexibility. The simpler version used here was chosen in order to reduce the complexity of assertions and therefore increase the efficiency of the SMT solver judgements. It is simpler because the parameter type list is flattened out so that the type of the set representing method method metadata uses only integer and string types in the tuple, and doesn't contain a nested set.

The reflective procedures detailed here have been sufficient for the examples undertaken to date, however it is by no means complete in comparison to the Java reflection API. There is some scope for extending the library with more features, for instance with primitive string and array types, or constructors with arguments. Additionally, it is possible for more object-oriented programming features to be supported, such as inheritence and access modifiers.

Other future work could be considering whether it would be possible to further increase automation. One such case would be the direct accessing of field contents when not using the reflection. Due to the fields being described by an inductive predicate, the verification system needs to know what and when to unfold predicates. For example, accessing the fourth field of an object will necessitate four unfoldings of $ObjFields predicates, in addition to unfolding the $Object. Then, once the field has been accessed, the same predicates must be re-folded. Every fold and unfold requires a hint annotation, and obviously every

such field access will follow a very similar pattern. It is not difficult to envisage some automation here where a field access can be recognised and the appropriate number of unfolds and folds can be automatically undertaken based on the field offset. An analogous approach could also be used for updating fields.

Recall that not all methods are included in the metadata as being reflectively invokable, because they do not fulfil the $Pre and $Post definitions. This has not been a problem so far, as no examples have yet been verified where it is desired to reflect upon methods without necessarily reflectively invoking them. However it would probably be beneficial to allow such behaviour. For facilitating such circumstances, where a method *may or may not* be invokable, it seems that there can be a couple of options. First, add an additional disjunct to the $MethodLseg definition, which is a copy of the non-empty case seen earlier, except the code is removed. Alternatively, extend the $Pre and $Post definitions to each include a case for unrecognised methods, i.e. by including a disjunct *false* in each definition.

Chapter 4

Reflection case studies

In this chapter two example uses of reflection are presented, which can be verified using the features and support for reflection obtained from Chapters 2 and 3. Both the examples are cases of generic-style programs, based on real examples found in Java developer resources. As such, the underlying presumption is that the Java examples have been transformed into the Crowfoot language. The examples are useful both in establishing the utility of the system for verifying reflective programs, along with demonstrating by-example the manner in which the approach from the previous chapter is to be applied.

The first example, in Section 4.1, shows a technique for verifying a reflective version of the visitor pattern, and for verifying a simple instance where the data structure is a binary tree. This example uses reflection to identify the type of the node being visited, and dispatch to the relevant handler in the visitor. The latter part requires the use of the invoke member of the Method metaclass, and so it is necessary to provide definitions for the $Pre and $Post predicates that were only universally quantified when the reflective library was presented.

The second example, in Section 4.2, gives an example of an implementation of serialization and deserialization algorithms, sometimes known as (un-)marshalling, where an object and its descendants can be converted to or from some string-based structure. The processes here hinge on being able to access and update the content of an object's fields generically for objects of any class. Additionally, the reflective construction of new objects is crucial for deserialization.

4.1 Reflective visitor pattern

4.1.1 Introduction

The visitor pattern [71] provides a general method for traversing a data structure where operations are performed on each element, with the operation possibly dependent on the type of the object being visited. The pattern separates the traversal algorithm from the operations, which means that it is straight-forward to add a new operation on the structure in that it doesn't require altering the code of data structure. In essence, the pattern

```
interface Visitor {
  visitParent(Parent p);
  visitLeaf(Leaf l);
}
class CountVisitor implements Visitor {
  int val = 0;
  visitParent(Parent p) {
    p.l.accept(this);
    p.r.accept(this);
  }
  visitLeaf(Leaf l) { val++; }
}
interface Node {
  accept(Visitor v);
}
class Parent implements Node {
  Node l,r;
  Parent(Object l, Object r) {
    this.l = l;
    this.r = r;
  }
  accept(Visitor v) { v.visitParent(this); }
}
class Leaf implements Node {
  int val;
  accept(Visitor v) { v.visitLeaf(this); }
}
```

Figure 4.1: Instance of the standard, non-reflective visitor pattern

provides a generic traversal that can be instantiated with different concrete algorithms; once the structure has been designed to fit the visitor *pattern* it will support any visitor.

A Java-like implementation of an instance of the visitor pattern is given in Figure 4.1. Here, the structure is a tree with Leaf and Parent nodes, and a concrete visitor CountVisitor whose behaviour is to count the number of leaves. Adding another operation on the tree structure is a case of creating a new class implementing the Visitor interface.

The standard version of the visitor pattern does still have disadvantages. For instance, if the shape of the data structure is altered, then the visitor interface will need to be updated, along with every concrete visitor. This does not fit the object-oriented programming principle of a program being open to extension and closed to modification [72]. Another disadvantage is that the *accept* methods introduce an indirection, which makes a program more complex to read and write, and demonstrates the intrusiveness of the design pattern because the traversal is not simply an extension: rather it is deeply embedded into the data structure.

The reflective visitor pattern [50][73] is an alternative approach that harnesses the fea-

```
abstract class Visitor {
  visit(Object obj) {
    try {
      Class obj_class = obj.getClass();
      String obj_className = obj_class.getName();
      String methodName = "visit" + obj_className;
      Class this_class = this.getClass();
      Method method = this_class.getMethod(methodName, obj_class);
      method.invoke(this, obj);
    }
    catch (NoSuchMethodException e) { }
    catch (IllegalAccessException e) { }
    catch (InvocationTargetException e) { }
  }
}
class CountVisitor extends Visitor {
  int val = 0;
  public void visitParent(Parent p) { visit(p.l); visit(p.r); }
  public void visitLeaf(Leaf l) { val++; }
  public int getVal() { return val; }
}

class Parent {
  Object l,r;
  Parent(Object l, Object r) {
    this.l = l;
    this.r = r;
  }
}
class Leaf {
  int val;
  Leaf() { val = 0 ; }
}
```

Figure 4.2: Our reflective visitor instance

tures of reflection to increase the level of genericity with the added benefit of also tackling the above disadvantages of the standard version. The overall idea is to create a generic "visit" method which uses reflection to decide to which *concrete* visit method to delegate based on the dynamic type of the node being visited. This removes the need for the accept methods, and thus any need for the data structures' classes to have any awareness of the visitors.

An instance of the reflective version, again in Java-like syntax, can be seen in Figure 4.2. The Visitor interface has been replaced by an abstract class, with just one concrete method "visit". This method is designed to handle any type of object, but thanks to the reflection the code is not complex. The first two commands retrieve the string name of the class of

the object to be visited. Next, that name is appended to the string "visit", the result of which is then used to look up the method of that name in the current (visitor) object's class. The method is then invoked, giving as argument the current object (the visitor) and the object being visited (a node). Java makes use of exceptions to avoid runtime crashes, which must be caught. The first, NoSuchMethodException is thrown by getMethod if no method of the given name exists, which in this case means visiting an object of an unexpected class. The final two exceptions are thrown by the invoke, if the method is not accessible from the target object (e.g. it is private), or if the method being invoked (here visitParent/visitLeaf) throws an exception itself. It is desirable in this research to verify that the cases described by the exceptions will be avoided for a given program. Note that reflection only needs to be used in the visit method. The rest of the code is reflection-free.

In terms of the tree data structure, the Node interface can be removed because the accept methods are no longer required as the dispatching to the relevant visit method will be done by *visit*. The accept methods are therefore removed from the two node classes, which remain otherwise unchanged.

In this chapter, the reflective visitor pattern is translated to the Crowfoot language, which is briefly discussed in Section 4.1.2. Then the approach used to specify the example is presented in Section 4.1.3.

4.1.2 The program

To verify the visitor pattern example, the code is now written in the Crowfoot language using the reflective library presented in Chapter 3. With the exception of the necessary try-catch statements of Java and lack of return values in Crowfoot, the translation is straightforward. The translated version can be seen in Figure 4.3.

Recall that the metadata generation in Chapter 3 relies on the provision of constructors for any class that should be included. These must be nullary, and named "Class_construct". This explains the additional Parent constructor, with the two-argument constructor from the Java version still present with a different name. Also remember the way objects are stored on the heap, with extant fields in pairs with the first part being a flag to signal presence of a field. Therefore the fields are accessed by offsets starting at 2 and incremented by 2 each time, as seen in the body of visitParent where the left and right child fields are dereferenced.

The generic visit procedure is given separately in Figure 4.4. Due to the lack of support for inheritance, the translation from the Java version requires the visit method procedure to be explicitly included with every concrete visitor as it has been with CountVisitor. Whilst this is inconvenient, it needs only a trivial copy step if there were some automated translation taking place. As will be discussed in the next section, the specifications of visit for each different concrete visitor will be identical except for the occurrence of the class type of the target object, further demonstrating that the lack of inheritance here is not much of a hindrance. The body of visit is as the Java-like version, except with the

```
proc CountVisitor_visitLeaf(this, l) {
    locals currentVal;
    currentVal := [this + 2];
    [this + 2] := currentVal + 1
}

proc CountVisitor_visitParent(this, p) {
    locals left, right;
    // Visit left subtree
    left := [p + 2];
    call CountVisitor_visit(this, left);

    // Visit right subtree
    right := [p + 4];
    call CountVisitor_visit(this, right)
}
proc CountVisitor_construct(res) {
    locals obj;
    obj := new "CountVisitor", 1, 0, 0;
    [res] := obj
}

proc CountVisitor_getVal(this, res) {
    [res] := [this + 1]
}

proc Parent_construct2(l, r, res) {
    locals obj;
    obj := new "Parent", 1, l, 1, r, 0;
    [res] := obj
}

proc Parent_construct(res) {
    locals l, r, obj;
    call Leaf_construct(res);
    l := [res];
    call Leaf_construct(res);
    r := [res];
    obj := new "Parent", 1, l, 1, r, 0;
    [res] := obj
}

proc Leaf_construct(res) {
    locals obj;
    obj := new "Leaf", 1, 0, 0;
    [res] := obj
}
```

Figure 4.3: Reflective visitor pattern example implemented in Crowfoot

addition of the *res* pointer used to handle return values and the removal of the try-catch statements which will be proved to be avoided.

4.1.3 Specification and verification

In order to verify the example, all the procedures need to be given specifications. To assist in describing the required behaviour, several predicates will be defined in addition to providing concrete definitions of the $Pre and $Post predicates necessary to utilize the reflective library's invoke procedure.

The definitions for the new predicates of the example can be seen in Figure 4.5. The first predicate describes the concrete tree data structure that is employed in this example, i.e. a tree with two node types, Leaf and Parent. The predicate contains two disjuncts, and describes a tree for some root object at address *root* of class @*type* with all elements

```
proc CountVisitor_visit(this, obj) {
  locals res, obj_class, @obj_className, this_class, @methodName, method;
  res := new 0;
  call Object_getClass(obj, res);
  obj_class := [res];
  call Class_getName(obj_class, res);
  @obj_className := [res];
  @methodName := "visit" ^ @obj_className;
  call Object_getClass(this, res);
  this_class := [res];
  call Class_getMethod(this_class, @methodName, res);
  method := [res];
  call Method_invoke(method, this, obj);
  dispose res
}
```

Figure 4.4: A visitor using reflective features, presented in the Crowfoot language

of the tree represented in the set $\%T$. The simplest is the case for a node of the leaf class, which is a terminal element of the tree. In this case the predicate serves as a wrapper for a Leaf object, and the set $\%T$ holds the singleton element representing that object. The second disjunct contains a Parent object, where there are constraints on the fields that describe a left and right field each containing a subtree. The set $\%T$ therefore contains the current root object, plus the left and right subtrees. Additionally, it is stated that the root object's address is unique in $\%T$, and that the left and right subtrees do not overlap. Those two constraints are implicit from the definitions of $Tree and $Object, however they are included to make them explicit to save deeper unfolding. The use of $\in$ for the two constraints on the fields $\%fs$ is sufficiently weak such that Parent nodes may still contain other fields which aren't important for the visitor, although the simple example being used here does not have any additional fields. The predicate $Tree is therefore a collection of objects, much in the same way as $References from Section 3.2. However it is a more precise definition tailored for the specific example, and only requires the inclusion of the fields that will actually be used rather than the full closure of referenced objects.

The predicate $VisitMethods is designed to ensure that for the set of class names ($\%names$), there exists a method in the given metadata method list ($\%mList$) with a name matching the appension of "visit" to each of those class names.

The last two predicates describe the functional behaviour of an invocation of one of the visit procedures. In this example instance, the behaviour should be that the value field in the visitor, which is its only field, is equal to the number of leaves in the tree data structure. The $\%pre$ and $\%post$ arguments represent the state of the fields before and

$$
\begin{array}{l}
\mathsf{recdef}\ \$\mathrm{Tree}(\mathit{root}; \%T; @\mathit{type}) := \\
\quad \exists\ \%\mathit{fs}, \%L, \%R, l, @l\_\mathit{type}, r, @r\_\mathit{type}. \\
\qquad @\mathit{type} = \text{“}\texttt{Parent}\text{”} \star \$\mathrm{Object}(\mathit{root}; \%\mathit{fs}; \text{“}\texttt{Parent}\text{”}) \star \mathit{root} \notin \mathrm{proj}_1(\%L \cup \%R) \star \\
\qquad \%T = \{(\mathit{root}, \text{“}\texttt{Parent}\text{”})\} \cup \%L \cup \%R \star \mathrm{proj}_1(\%L) \cap \mathrm{proj}_1(\%R) = \emptyset \star \\
\qquad (\mathit{root}+1, @l\_\mathit{type}, l) \in \%\mathit{fs} \star \$\mathrm{Tree}(l; \%L; @l\_\mathit{type}) \star \\
\qquad (\mathit{root}+3, @r\_\mathit{type}, r) \in \%\mathit{fs} \star \$\mathrm{Tree}(r; \%R; @r\_\mathit{type}) \\
\vee\ \exists\ \%\mathit{fs}.\ @\mathit{type} = \text{“}\texttt{Leaf}\text{”} \star \$\mathrm{Object}(\mathit{root}; \%\mathit{fs}; \text{“}\texttt{Leaf}\text{”}) \star \%T = \{(\mathit{root}, \text{“}\texttt{Leaf}\text{”})\};
\end{array}
$$

$$
\begin{array}{l}
\mathsf{recdef}\ \$\mathrm{VisitMethods}(; \%\mathit{types}, \%\mathit{mList}) := \%\mathit{types} = \emptyset \\
\vee\ \exists\ \%\mathit{rest}, a, @\mathit{type}. \\
\qquad \%\mathit{types} = \{@\mathit{type}\} \cup \%\mathit{rest} \star @\mathit{type} \notin \%\mathit{rest} \star \\
\qquad (a, \text{“}\texttt{visit}\text{”} +\!\!+ @\mathit{type}, \text{“}\texttt{CountVisitor}\text{”}, @\mathit{type}) \in \%\mathit{mList} \star \\
\qquad \$\mathrm{VisitMethods}(; \%\mathit{rest}, \%\mathit{mList});
\end{array}
$$

$$
\begin{array}{l}
\mathsf{recdef}\ \$\mathrm{Fun}(; \%\mathit{struct}, \%\mathit{pre}, \%\mathit{post}) := \\
\quad \exists\ o, m, n.\ \$\mathrm{NumLeaves}(n; \%\mathit{struct}) \star \\
\qquad \%\mathit{pre} = \{(o+1, \text{“}\texttt{int}\text{”}, m)\} \star \%\mathit{post} = \{(o+1, \text{“}\texttt{int}\text{”}, m+n)\};
\end{array}
$$

$$
\begin{array}{l}
\mathsf{recdef}\ \$\mathrm{NumLeaves}(n; \%T) := n = 0 \star \%T = \emptyset \\
\vee\ \exists\ @\mathit{type}, \mathit{root}, \%\mathit{rest}. \\
\qquad @\mathit{type} = \text{“}\texttt{Leaf}\text{”} \star \%T = \{(\mathit{root}, @\mathit{type})\} \cup \%\mathit{rest} \star \mathit{root} \notin \mathrm{proj}_1(\%\mathit{rest}) \star \\
\qquad \text{“}\texttt{Parent}\text{”} \notin \mathrm{proj}_2(\%\mathit{rest}) \star \$\mathrm{NumLeaves}(n-1; \%\mathit{rest}) \\
\vee\ \exists\ @\mathit{type}, \mathit{root}, \%\mathit{rest}. \\
\qquad @\mathit{type} = \text{“}\texttt{Parent}\text{”} \star \%T = \{(\mathit{root}, @\mathit{type})\} \cup \%\mathit{rest} \star \mathit{root} \notin \mathrm{proj}_1(\%\mathit{rest}) \star \\
\qquad \$\mathrm{NumLeaves}(n; \%\mathit{rest});
\end{array}
$$

Figure 4.5: Predicates used for describing visitor pattern behaviour

after the execution. The value of the field afterwards is therefore the previous value plus the size of the tree structure given in the first argument. The tree size is calculated by the $NumLeaves predicate which is defined inductively with two non-empty cases: the first is for a Leaf node which increments the size counter n, and the second is for a Parent node which maintains the same counter value. The definition is standard except for the Leaf case, which includes an additional constraint ($\text{“}\texttt{Parent}\text{”} \notin \mathrm{proj}_2(\%\mathit{rest})$). The purpose of this is to force determinism when unfolding the predicate, such that all parent notes are removed first before leaf nodes can be removed.

The definitions for the $Pre and $Post predicates, which will be used to specify the behaviour of a reflectively invoked visitX method, are shown in Figure 4.6. Both the pre- and post-condition include the spatial information regarding the target object (a visitor), the tree structure being visited with root *arg*, and the metadata. The metadata is impor-

$$\begin{array}{l}\mathsf{recdef}\ \$\mathrm{Pre}(target, arg; \%targFsPre, \%M, \%T; @targetClass, @mName, @argClass) := \\ \quad \exists\, targetClassId, \%clsFs, construct, \%methodNames, \%vMs. \\ \qquad @targetClass = \text{“}\texttt{CountVisitor}\text{”} \star \%targFsPre = \{(target + 1, \text{“}\texttt{int}\text{”}, n)\} \\ \qquad \star\, \$\mathrm{Object}(target; \%targFsPre; @targetClass) \\ \qquad \star\, \$\mathrm{Tree}(arg; \%T; @argClass) \\ \qquad \star\, \$\mathrm{Meta}(; \%M) \star \mathrm{proj}_2(\%T) \subseteq \mathrm{proj}_2(\%M) \\ \qquad \star\, (targetClassId, @targetClass, \%clsFs, \%vMs, construct) \in \%M \\ \qquad \star\, \$\mathrm{VisitMethods}(; \mathrm{proj}_2(\%T), \%vMs) \star @mName = \text{“}\texttt{visit}\text{”} +\!\!+ @argClass;\end{array}$$

$$\begin{array}{l}\mathsf{recdef}\ \$\mathrm{Post}(target, arg; \%targFsPre, \%M, \%T; @targetClass, @mName, @argClass) := \\ \quad \exists\, \%tFsPost. \\ \qquad \$\mathrm{Object}(target; \%tFsPost; @targetClass) \\ \qquad \star\, \$\mathrm{Tree}(arg; \%T; @argClass) \\ \qquad \star\, \$\mathrm{Meta}(; \%M) \\ \qquad \star\, \$\mathrm{Fun}(; \%T, \%targFsPre, \%tFsPost);\end{array}$$

Figure 4.6: Concrete definitions for $Pre and $Post predicates used by reflective invoke

tant because although the concrete visit methods do not use reflection directly themselves, they may call the generic visit method to continue the traversal as in visitParent. The rest of the pre-condition is pure and states that: all the classes of nodes appearing in the tree are included in the metadata, the target (visitor) class appears in the metadata, and that there exists a method “visitX” for every tree node class (see $VisitMethods definition in Figure 4.5).

The post-condition shows that the fields of the target (visitor) have been updated, and they are related to the original fields by the instance of the predicate $Fun.

It is useful to highlight that the $Pre and $Post definitions are designed to be somewhat generic with respect to accommodating different visitors. That is to say that the program may use other visitors, performing different operations than the simple Leaf count, without needing to alter the $Pre and $Post definitions. Instead the different behaviour is effectively parametrized by the $Fun predicate which can be altered as required.

4.1.3.1 Specifications

The fully annotated version of the visitLeaf method in the the Crowfoot language is given in Figure 4.7. This includes the specification and “hints” to the automated prover in pale font. The specification of visitLeaf must entail the generic method specification in order for it to be used reflectively. This method is an example where the concrete specification is smaller/weaker than the generic one, and so the frame rule is required to complete the entailment. Of course, the explicit $Pre and $Post predicates could be used and the

```
proc CountVisitor_visitLeaf(this, l)
∀%T, v.
pre : $Object(this; {(this + 1, "int", v)}; "CountVisitor") ⋆ $Tree(l; %T; "Leaf");
post : ∃%fsPost. $Object(this; %fsPost; "CountVisitor") ⋆ $Tree(l; %T; "Leaf") ⋆
        $Fun(; %T, {(this + 1, "int", v)}, %fsPost);
{
  locals currentVal;
  ghost 'unfold $Object(this; ?; "CountVisitor")';
  ghost 'unfold $ObjFields(this + 1; ?)';
  currentVal := [this + 2];
  [this + 2] := currentVal + 1;
  // Show that there are no other fields
  ghost 'unfold $ObjFields(this + 3; ?)'; ghost "fold $ObjFields(this + 3; ?)';

  ghost 'fold $ObjFields(this + 1; {(this + 1, "int", v + 1)})';
  ghost 'fold $Object(this; {(this + 1, "int", v + 1)}; "CountVisitor")';

  // Prove that the function (in $Fun) has been implemented
  ghost 'unfold $Tree(l; %T; "Leaf")';
  ghost 'fold $NumLeaves(0; ∅)';
  ghost 'fold $NumLeaves(1; {(l, "Leaf")})';
  ghost 'fold $Tree(l; %T; "Leaf")';
  ghost 'fold $Fun(; {(l, "Leaf")}, {(this + 1, "int", v)}, {(this + 1, "int", v + 1)})'
}
```

Figure 4.7: Leaf visitor implemented in Crowfoot, including specification and annotations

additional formulae would be ignored, however this goes against the modularity principles of separation logic. The pre-condition requires firstly that the target object is on the heap, which has type CountVisitor and a single field with value v. Secondly, it needs the (sub-)tree being visited, with the root given by the procedure's second argument, and the type of the root being Leaf. The post-condition includes the target object with the updated field, the same shape tree, and asserts the relation between the visitor's original field and the new field. The frame that will be automatically discovered to fulfil the defintions in Figures 4.6 is:

$$\begin{aligned}&\$\mathrm{Meta}(;\%M) \star \mathrm{proj}_2(\%T) \subseteq \mathrm{proj}_2(\%M) \star \\ &(cPtr, \text{“}\texttt{CountVisitor}\text{”}, \%cFs, \%cMs, cons) \in \%M \star \\ &\$\mathrm{VisitMethods}(;\mathrm{proj}_2(\%T), \%cMs)\end{aligned}$$

proc $CountVisitor\_visitParent(this, p)$
$\forall \%fsPre, \%M, \%T, @l\_type, l, @r\_type.$
pre : \$Pre($this, p; \%fsPre, \%M, \%T$;"`CountVisitor`","`visitParent`","`Parent`");
post : \$Post($this, p; \%fsPre, \%M, \%T$;"`CountVisitor`","`visitParent`","`Parent`");

Figure 4.8: Specification of visitParent

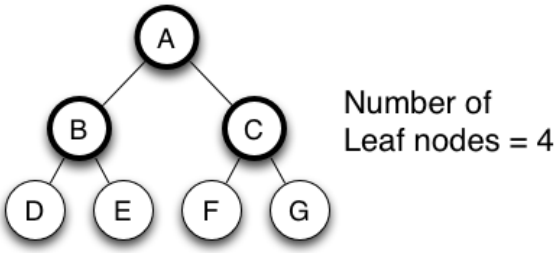
for some fresh variables $\{\%M, cPtr, \%cFs, \%cMs, cons\}$. The definition of \$Post only requires \$Meta, but entailments between specifications allow weakening of a post-condition.

The specification for visitParent, in Figure 4.8, almost trivially entails the generic specification by explicitly including exactly the \$Pre and \$Post predicates. The work in showing the entailment when generating the metadata is therefore reduced to just variable instantiations, which is handled by the entailment prover. The specification of the generic visit in Figure 4.9 also consists solely of the \$Pre and \$Post predicates, differing only in that the target fields and argument object class type have been abstracted to variables. Given that this visit method will (reflectively) invoke the concrete visitLeaf/visitParent methods, it is unsurprising that its specification represents a superset of their specifications (although this generic visit will never be reflectively invoked itself directly). Note the use of lemmas towards the end. The first performs an unfolding of the \$VisitMethods predicate, and the second exposes the implicit fact that all method names in the metadata are unique.

Because the fields are not manipulated reflectively, one need not worry about making the constructor's specifications such that the field types are able to be inferred, as required by the *generateMeta* function in Chapter 3. Instead, the field list in the metadata will be empty. This flexibility keeps the specifications simpler through the possibility of more bespoke assertions that describe the specific situation. In this case, it means we can model the very specific case of subclassing for the Tree nodes, which is not otherwise supported.

4.1.4 Using the program

Whilst the main focus of the case study here has been to illustrate the verification of the reflective aspects of the programs, to complete the demonstration a small and contrived example that uses the previously described visitor has also been created. The program creates a simple binary tree, consisting of three Parent nodes and four Leaf nodes:



```
proc CountVisitor_visit(this, obj)
  ∀ %fsPre, %M, %T, @type, @mName.
  pre : $Pre(this, obj; %fsPre, %M, %T; "CountVisitor", @mName, @type);
  post : $Post(this, obj; %fsPre, %M, %T; "CountVisitor", @mName, @type);
{
  locals res, obj_class, @obj_className, this_class, @methodName, method;
  res := new 0;
  ghost "unfold $Pre(this, obj; %fsPre, %M, %T; "CountVisitor", @mName, @type)";
  // Get this class
  call Object_getClass(this, res);
  this_class := [res];
  ghost "unfold $Tree(obj; %T; @type)";
  // Get obj class
  call Object_getClass(obj, res);
  obj_class := [res];
  // Get obj class name
  call Class_getName(obj_class, res);
  @obj_className := [res];
  // Append "visit"
  @methodName := "visit"++@obj_className;
  ghost "lemma unfold_methodsExist()";

  // Get method in this
  [res] := 0;
  call Class_getMethod(this_class, @methodName, res);
  method := [res];

  // Invoke method on this with arg: obj
  ghost "fold $Tree(obj; %T; @type)";
  ghost "lemma unfold_visitMethods()";
  ghost "lemma method_name_unique()";
  ghost "fold $Pre(this, obj; %fsPre, %M, %T; "CountVisitor", @mName, @type)";
  call Method_invoke(method, this, obj, res);
  dispose res
}
```

Figure 4.9: Implementation and specification of the generic visit

A leaf-counting visitor is then created and deployed which is expected to therefore result in recording that it has seen four leaves. The (annotated) code which accomplishes this is given in Figure 4.10.

The pre-condition must contain the trigger to ensure that the metadata is created. Additionally there is a result pointer which will be used throughout the program and to store the count result given by the visitor, which is assured to be four in the post-condition. The body first declares some local variables for each object, along with an extra for representing the constant "1" because arguments to call/eval operations may not be literals. Following this, the tree is set up by first declaring all the leaves (D to G), followed by the three parents. The arguments of the parent nodes' constructors respectively construct the tree of the desired shape. The lemmas expose the hidden uniqueness of the object addresses, which is important for unfolding the tree. Next the visitor is created.

Before the visiting process can begin, the symbolic state must be shown to meet the conditions required, including presence of all required visitX methods and well-definedness of object in the metadata. An instance of $VisitMethods is created in three steps: first for the empty case, then it is built up with the two class types. The second argument of this predicate is the list of methods in the metadata, where the $m?$ is some unknown fresh variable allocated during the metadata generation. The value of this argument must be provided for the initial empty case, but the later folds will be able to infer it by the arguments of the inductive occurrence. Finally, with this predicate created and with the metadata automatically created on the heap, it is possible to show the desired $Pre.

After calling the visit method the $Pre instance will have been replaced by an instance of $Post, which will contain the predicate describing the function of the visitor. With this information and by accessing the visitor's counter field it can be shown that the result was four, as to show the post-condition is fulfilled. The *check_four* lemma simply unfolds the $NumLeaves predicate until it reaches the empty-case where the size is zero.

```
proc main(res)
  pre : meta ↦ _ ⋆ res ↦ _;
  post : ∃%M. $Meta(;%M) ⋆ res ↦ 4;
{
  locals A, B, C, D, E, F, G, visitor, one;  one := 1;

  // Build a test tree
  // We'll set all the leaves' value fields to 1, we only ever count the nodes anyway
  call Leaf_construct(one, res);  G := [res];
  call Leaf_construct(one, res);  F := [res];
  call Leaf_construct(one, res);  E := [res];
  call Leaf_construct(one, res);  D := [res];
  call Parent_construct(F, G, res);  C := [res];
  call Parent_construct(D, E, res);  B := [res];
  call Parent_construct(B, C, res);  A := [res];
  ghost 'lemma unique_tree_objs()';

  // Make a new visitor
  call CountVisitor__construct(res);
  visitor := [res];

  // Let's start visiting!
  ghost 'fold $VisitMethods(;∅, {(m3,"visitParent","CountVisitor","Parent")}
                                ∪{(m2,"visitLeaf","CountVisitor","Leaf")}
                                ∪{(m1,"visit","CountVisitor",@type11)})';
  ghost 'fold $VisitMethods(;{"Parent"},?)';
  ghost 'fold $VisitMethods(;{"Leaf"} ∪ {"Parent"},?)';
  ghost 'fold $Pre(visitor, A;?,?,?;"CountVisitor","Parent")';
  call CountVisitor_visit(visitor, A);
  ghost 'unfold $Post(visitor, A;?,?,?;?,?)';  ghost 'unfold $Fun(;?,?,?)';
  // Take the result (and check it is 4)
  call CountVisitor_getVal(visitor, res);
  ghost 'lemma check_four()';

  // Cleanup
  call CountVisitor__dispose(visitor);  call Parent__dispose(A)
}
```

Figure 4.10: Main procedure which constructs and visits a tree

4.2 Reflective serialization/deserialization

4.2.1 Introduction

Serialization, known in some settings as marshalling, is a method for translating the running state of a resource (such as an object) within a program into some data format (usually textual) for persistent storage. For example an object, including its current field values, can be serialized by some encoding into a simple text file. There are a number of uses for serialization, primarily concerned with the communication of objects over networks, which is useful for remote procedure call applications. Another use can be to save the current state of an object so that it may be compared to a later version of the object after some operation has taken place, to see whether the operation had any effect. Similarly, the persistent copy can be used for rolling back to an earlier state during a failed transaction process.

Due to serialization being concerned with the *current state* of an object, reflection is the natural programming feature that will be required for a serialization algorithm. Whilst it is possible to perform the required steps without reflection if the program structure is available in advance and classes and the individual fields are known, this would be a large amount of code hard-coded to certain classes, and the algorithm would be duplicated for each class. The reflection allows for a single algorithm that can handle any type of class. The algorithm chosen for serialization is taken from a Java Reflection textbook [45]. In a high-level sense, the serialization algorithm accepts an object, and creates an XML record for it, giving it a unique identifier. Next the algorithm proceeds recursively through the fields, and serializes each one. In the case of non-primitive (object) fields, the containing object's XML record keeps a reference to the unique identifier. Several simplifications have been made to the original implementation as given in [45]:

1. The XML structure is simplified by making it specific to the example of serialization. Ordinarily XML is generic.
2. Security features of Java are ignored, such that all fields are considered public. In the original, private fields first have to be converted to accessible.
3. The only primitive type supported are integers. Fields of other types must then be some class of object.
4. The while-loops have been replaced with recursion.

The specialization of the XML structure has been done to ease the process of giving types to sets for the pure reasoning performed by the SMT-solver. If the XML were generic, the representing sets would need to be recursively typed as an element can have any number of children. In this case, however, the structure is always the same and so the stronger specifications can be used.

Part of the reflective library that is required here that was not used for the visitor pattern example is the support for reflection on fields. Additionally, when it comes to deserializing an object it is necessary to reflectively construct new objects from the encoding. However, unlike the last example there is no need for reflection on methods here.

```
<serialized>
  <object class="A" id="0">
    <field name="x" declaringclass="A">
      <reference>1</reference>
    </field>
    <field name="y" declaringclass="A">
      <value>10</value>
    </field>
  </object>
  <object class="B" id="1">
    <field name="a" declaringclass="B">
      <value>4</value>
    </field>
    <field name="b" declaringclass="B">
      <value>1</value>
    </field>
  </object>
</serialized>
```

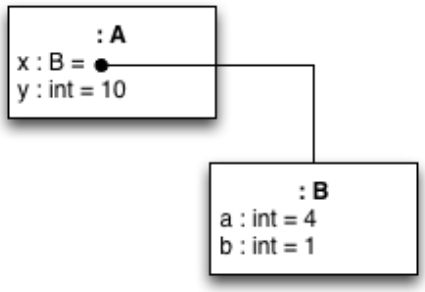


Figure 4.11: Example showing the structure of the XML data representing a serialized group of objects

In Section 4.2.2, the translated versions of classes that support the algorithm are presented, along with their specifications. This is the XML structure, instances of hash maps used to keep track of unique identifiers assigned to objects, and a wrapper class for the primitive integer type. Then in Section 4.2.3 the implementation of the serialization algorithm is presented, followed by the specifications. The algorithm for the deserialization is then given in Section 4.2.4.

4.2.2 Supporting program features

4.2.2.1 XML structure

As mentioned above, due to the lack of recursively typed sets the XML *structure* is fixed based on the shape required in this example. An example of what the XML structure might look like when stored as a text file is given in Figure 4.11. The example is small, containing two objects with two fields each. The "root" object (id 0) references the second object through its first field. The rest of the fields are primitive integers. It is clear that there will only be a fixed level of nesting (object → field → reference/value), which allows the provision of specifications without recursive set types. This simplification also allows the code using the XML to be simpler: previously all attributes had to be added to an element after creation, but because each type of element is fixed the attributes can be set during construction. This means the focus of the verification can remain on the use of reflection, rather than the auxiliary XML support.

Each element type (object, field, reference/value) is given its own constructor and set of methods, along with a predicate declaration for use in specifications. The suffixes of

$$\textsf{recdef}\ \$\mathrm{XMLDocument}(ptr; \%objs) := \exists\, objs.\ ptr \mapsto objs \star \$\mathrm{XMLObject}(objs; \%objs);$$

$$\begin{aligned}
&\textsf{recdef}\ \$\mathrm{XMLObject}(ptr; \%objs) := ptr = 0 \star \%objs = \emptyset \\
&\quad \vee \quad \exists\, @class, @id, fields, \%fs, next, \%rest. \\
&\quad ptr \mapsto @class, @id, fields, next \star \$\mathrm{XMLField}(fields; \%fs) \\
&\quad \star\ \$\mathrm{XMLObject}(next; \%rest) \star \%objs = \{(ptr, @id, @class, \%fs)\} \cup \%rest \\
&\quad \star\ ptr \notin \mathrm{proj}_1(\%rest) \star @id \notin proj(2, \%rest) \\
&\quad \star\ \mathrm{proj}_3(\%fs) \subseteq \{@class\} \star @class \neq \text{“}\texttt{int}\text{”};
\end{aligned}$$

$$\begin{aligned}
&\textsf{recdef}\ \$\mathrm{XMLField}(ptr; \%fs) := ptr = 0 \star \%fs = \emptyset \\
&\quad \vee \quad \exists\, @name, @decclass, child, @childText, @childType, next, \%rest. \\
&\quad ptr \mapsto @name, @decclass, child, next \star \$\mathrm{XMLRefVal}(child; ; @childType, @childText) \\
&\quad \star\ \$\mathrm{XMLField}(next; \%rest) \\
&\quad \star\ \%fs = \{(ptr, @name, @decclass, @childType, @childText)\} \cup \%rest \\
&\quad \star\, ptr \notin \mathrm{proj}_1(\%rest) \star @name \notin proj(2, \%rest);
\end{aligned}$$

$$\begin{aligned}
&\textsf{recdef}\ \$\mathrm{XMLRefVal}(ptr; ; @name, @text) := \\
&\quad ptr \mapsto @name, @text \star @name \in \{\text{“}\texttt{value}\text{”}\} \cup \{\text{“}\texttt{reference}\text{”}\};
\end{aligned}$$

Figure 4.12: Predicates describing an XML-like structure on the heap

the procedures all correspond to the equivalent method members of JDOM's [74] Element class[1], used by the algorithm given by Forman and Forman [45]. The predicate definitions representing instances of the different type of element are given in Figure 4.12. In essence, they are linked lists enriched with some additional element uniqueness properties, with the exception of $XMLRefVal which will always appear as a singleton child of a field (a field cannot have more than one value). Predicate definition $XMLObject also ensures that the third projection of the fields (declaring class) will be the same as the class of the owning object record. Predicate $XMLRefVal describes an element that must be named either "reference" or "value".

The specifications of procedures providing for the creation of an XML data structure are in Figure 4.13. For presentation the implementations have been omitted. The XMLDocument type is designed to represent the top-level element (in the example, `<serialized>`). In practice, because it is a singleton, it serves no real function other than a wrapper for an object list. The procedure *XMLDocument_addContent* allows the addition of a new object to an existing Document, as long as no identifiers overlap (the second projection of the object tuple). This is realized as a list append, although in practice the second object

[1] http://jdom.org/docs/apidocs/org/jdom2/Element.html

proc abstract $XMLDocument\_construct(res)$
pre : $res \mapsto \_$;
post : $\exists\, doc.\ res \mapsto doc \star \$\mathrm{XMLDocument}(doc; \emptyset)$;

proc abstract $XMLDocument\_addContent(doc, elt)$
$\forall\, \%doc, \%new.$
pre : $\$\mathrm{XMLDocument}(doc; \%doc) \star \$\mathrm{XMLObject}(elt; \%new)$
$\star\, \mathrm{proj}_2(\%doc) \cap \mathrm{proj}_2(\%new) = \emptyset$;
post : $\$\mathrm{XMLDocument}(doc; \%doc \cup \%new)$;

proc abstract $XMLObject\_construct(@class, @id, res)$
pre : $res \mapsto \_ \star @class \neq$ "`int`";
post : $\exists\, e.\ res \mapsto e \star \$\mathrm{XMLObject}(e; \{(e, @id, @class, \emptyset)\})$;

proc abstract $XMLObject\_addContent(e, fs)$
$\forall\, \%objs, @id, @class, \%fs, \%newFs.$
pre : $\$\mathrm{XMLObject}(e; \%objs) \star (e, @id, @class, \%fs) \in \%objs$
$\star\, \$\mathrm{XMLField}(fs; \%newFs) \star \%newFs \neq \emptyset \star \mathrm{proj}_3(\%newFs) \subseteq \{@class\}$
$\star\, \mathrm{proj}_2(\%fs) \cap proj(2, \%newFs) = \emptyset$;
post :
$\$\mathrm{XMLObject}(e; \%objs \setminus \{(e, @id, @class, \%fs)\} \cup \{(e, @id, @class, \%fs \cup \%newFs)\})$;

proc abstract $XMLField\_construct(@name, @decclass, refval, res)$
$\forall\, @childType, @childText.$
pre : $res \mapsto \_ \star \$\mathrm{XMLRefVal}(refval; ; @childType, @childText)$;
post : $\exists\, e.res \mapsto e \star \$\mathrm{XMLField}(e; \{(e, @name, @decclass, @childType, @childText)\})$;

proc abstract $XMLRefVal\_construct(@name, @text, res)$
pre : $res \mapsto \_ \star @name \in \{$"`value`"$\} \cup \{$"`reference`"$\}$;
post : $\exists\, rv.res \mapsto rv \star \$\mathrm{XMLRefVal}(rv; ; @name, @text)$;

proc abstract $XMLRefVal\_getText(e, res)$
$\forall\, \%attrs, \%descs, @name, @text.$
pre : $\$\mathrm{XMLRefVal}(e; ; @name, @text) \star res \mapsto \_$;
post : $\$\mathrm{XMLRefVal}(e; ; @name, @text) \star res \mapsto @text$;

proc abstract $XMLRefVal\_getName(e, res)$
$\forall\, \%attrs, \%descs, @name, @text.$
pre : $\$\mathrm{XMLRefVal}(e; ; @name, @text) \star res \mapsto \_$;
post : $\$\mathrm{XMLRefVal}(e; ; @name, @text) \star res \mapsto @name$;

Figure 4.13: Procedures for manipulating the XML structure

list will always be a single element. For `<object>` elements, there is a constructor which also sets the ID and class attributes, and a procedure for adding children (fields). The *XMLObject_addContent* specification requires that the fields are admissible to the current set, in that there is no overlap of field names and that their declaring-class attribute matches the parent-to-be object's class. The post-condition uses set-minus to assert that the new object set is updated just for the single pertinent object. Note this is also more generic than required by the example, allowing the update of the fields in any object element in a list, but as will be seen in the example, the actual $XMLObject list will be a singleton.

For fields, the XMLField class of elements has just a constructor. The attribute and value accessors are trivial, requiring simply the unfolding of a predicate instance and a dereference operation, and have been omitted from the example in favour of the direct dereferencing.

The leaf of the XML hierarchy is the content of a field element, either `<value>` or `<reference>` which has a constructor and accessors for the name of the element and the content value.

4.2.2.2 Reference tables

The algorithms use two data-structures, based on Java's Map[2]. The first, for serialization, is a map indexed by objects that contains string identifier numbers. The second is the inverse, mapping identifiers to object pointers. To avoid the complexities of parametric types, two concrete versions of the map are provided. Because their implementation is not of interest for the purposes of the reflection reasoning, only the specifications are provided. These are given in Figures 4.14 and 4.15.

The specifications are the obvious ones, with the possible exception of additional constraints ensuring that when adding to the maps (with *_put*) there is not a member already present with the given key. Note that the *size* operation is only required for the serialization process, and therefore is only present in the IdentityHashMap. Predicates $\$\text{HashMap}(m; \%M)$ and $\$\text{IdMap}(m; \%M)$ describe the respective maps on the heap, starting at address m, with content described by the set $\%M$, where $\%M$ is a set of key-value pairs.

4.2.2.3 Integer wrapper class and string conversion

Due to the fact that the reflective *Field.set* operation expects an object, the primitives need to be wrapped up as objects in order for the deserialization algorithm to succeed. In this case, an int must become an Integer object. Therefore, in Figure 4.16 the Integer class is given with a constructor, and also some further methods to handle conversion between strings and integers[3].

[2] http://docs.oracle.com/javase/6/docs/api/java/util/Map.html

[3] Since Java 5, which was released after the Forman/Forman [45] publication, "boxing" conversion was introduced which automatically converts between primitives and wrappers as required.

$$
\begin{array}{l}
\mathsf{proc\ abstract}\ \mathit{HashMap\_construct}(\mathit{res}) \\
\quad \mathsf{pre} : \mathit{res} \mapsto \_; \\
\quad \mathsf{post} : \exists\, \mathit{tbl}.\ \mathit{res} \mapsto \mathit{tbl} \star \$\mathrm{HashMap}(\mathit{tbl}; \emptyset); \\
\mathsf{proc\ abstract}\ \mathit{HashMap\_put}(\mathit{table}, @\mathit{id}, \mathit{value}) \\
\quad \forall\, \%\mathit{data}. \\
\quad \mathsf{pre} : \$\mathrm{HashMap}(\mathit{table}; \%\mathit{data}) \star @\mathit{id} \notin \mathrm{proj}_1(\%\mathit{data}); \\
\quad \mathsf{post} : \$\mathrm{HashMap}(\mathit{table}; \%\mathit{data} \cup \{(@\mathit{id}, \mathit{value})\}); \\
\mathsf{proc\ abstract}\ \mathit{HashMap\_get}(\mathit{table}, @\mathit{id}, \mathit{res}) \\
\quad \forall\, \%\mathit{tbl}. \\
\quad \mathsf{pre} : \mathit{res} \mapsto \_ \star \$\mathrm{HashMap}(\mathit{table}; \%\mathit{tbl}) \star @\mathit{id} \in \mathrm{proj}_1(\%\mathit{tbl}); \\
\quad \mathsf{post} : \exists\, \mathit{value}, \%\mathit{rest}.\ \mathit{res} \mapsto \mathit{value} \star \$\mathrm{HashMap}(\mathit{table}; \%\mathit{tbl}) \\
\quad \star \{(@\mathit{id}, \mathit{value})\} \cup \%\mathit{rest} = \%\mathit{tbl} \star @\mathit{id} \notin \mathrm{proj}_1(\%\mathit{rest}); \\
\mathsf{proc\ abstract}\ \mathit{HashMap\_containsKey}(\mathit{table}, @\mathit{id}, \mathit{res}) \\
\quad \forall\, \%\mathit{tbl}. \\
\quad \mathsf{pre} : \$\mathrm{HashMap}(\mathit{table}; \%\mathit{tbl}) \star \mathit{res} \mapsto \_; \\
\quad \mathsf{post} : \$\mathrm{HashMap}(\mathit{table}; \%\mathit{tbl}) \star \mathit{res} \mapsto 1 \star @\mathit{id} \in \mathrm{proj}_1(\%\mathit{tbl}) \\
\quad \vee \quad \$\mathrm{HashMap}(\mathit{table}; \%\mathit{tbl}) \star \mathit{res} \mapsto 0 \star @\mathit{id} \notin \mathrm{proj}_1(\%\mathit{tbl});
\end{array}
$$

Figure 4.14: Map from identifiers (as strings) to object addresses (integers)

The standard constructor *Integer_construct* is straightforward, with the specification showing that Integer objects have a single field of primitive type int. It is also necessary to have an extraordinary constructor, *Integer_valueOf*, that allows the new Integer object to represent the integer contained in a string. To perform the conversion between strings and integers, for which there is no built-in support in the verification system, the details are left abstract. This is achieved through a universally quantified predicate, $Cast, supported by some lemmas which assert the crucial properties. An instance of $\$\mathrm{Cast}(n;;@s)$ is intended to infer that the string $@s$ can be parsed as the integer n, and vice versa. The two lemmas which describe the properties of the predicate are given in Figure 4.17. The first, *strint_injectivity* asserts that multiple conversions of the same integer will yield the same string value. The second, *strint_zero* gives a concrete case for converting the integer 0, which should be converted to the string "0".

In cooperation with this faux-support for conversion, there are two procedures defined from static methods in Java's Integer class, for setting up string-integer parsing in both directions. These must be abstract because they use the $Cast predicate, for which there is no definition. Using these, it is then possible to define the special constructor *Integer_valueOf*, mentioned above. The post-condition then includes the $Cast predicate to associate the field's value with the string argument.

proc abstract $IdentityHashMap\_construct(res)$
  pre : $res \mapsto \_$;
  post : $\exists\, m.\ res \mapsto m \star \$\mathrm{IDMap}(m, 0; \emptyset)$;
proc abstract $IdentityHashMap\_put(map, obj, @value)$
  $\forall\, \%M, size, \%fs, @type.$
  pre : $\$\mathrm{IDMap}(map, size; \%M) \star \$\mathrm{Object}(obj; \%fs; @type) \star obj \notin \mathrm{proj}_1(\%M)$;
  post : $\$\mathrm{IDMap}(map, size + 1; \%M \cup \{(obj, @value)\}) \star \$\mathrm{Object}(obj; \%fs; @type)$;
proc abstract $IdentityHashMap\_get(map, obj, res)$
  $\forall\, \%M, size.$
  pre : $res \mapsto \_ \star \$\mathrm{IDMap}(map, size; \%M) \star obj \in \mathrm{proj}_1(\%M)$;
  post : $\exists\, @value, \%rest.\ res \mapsto @value \star \$\mathrm{IDMap}(map, size; \%M)$
  $\star\, \{(obj, @value)\} \cup \%rest = \%M \star obj \notin \mathrm{proj}_1(\%rest)$;
proc abstract $IdentityHashMap\_containsKey(map, obj, res)$
  $\forall\, \%M, size.$
  pre : $\$\mathrm{IDMap}(map, size; \%M) \star res \mapsto \_$;
  post : $\$\mathrm{IDMap}(map, size; \%M) \star res \mapsto 1 \star obj \in \mathrm{proj}_1(\%M)$
  $\vee \quad \$\mathrm{IDMap}(map, size; \%M) \star res \mapsto 0 \star obj \notin \mathrm{proj}_1(\%M)$;
proc abstract $IdentityHashMap\_size(map, res)$
  $\forall\, \%M, size.$
  pre : $res \mapsto \_ \star \$\mathrm{IDMap}(map, size; \%M)$;
  post : $res \mapsto size \star \$\mathrm{IDMap}(map, size; \%M)$;

Figure 4.15: Map from objects (by unique integer addresses) to identifiers (as strings)

4.2.3 Serialization

4.2.3.1 The program

The first side of the problem, which is arguably the more simple of the two because the reflection is read-only, is the serialization of an object to the XML data structure. The algorithm makes use of the IdentityHashMap.

The algorithm is recursive, and starts by accepting an object which it adds to the XML and map. Next, each of the fields is serialized: if the field contains a primitive value (int) then it is simply added to the XML as a child of the parent object; if the field contains an object reference then that object is sought in the map, recursively serialized if not present, and the assigned identifier in the map is stored in the XML field data for the original parent object. By using the map, fields that reference the same object do not serialize into separate objects. The recursive nature can be seen by the call graph in Figure 4.18, which shows how *serializeVariable* re-enters the algorithm by calling *serializeHelper*. The connection is also indicative of the way that specifications for the procedures are highly

```
proc Integer_construct(val, res)
  pre : res ↦ _;
  post : ∃ o.res ↦ o ⋆ $References(; ∅, {(o, "Integer", {(o + 1, "int", val)})});
{
  locals obj;
  obj := new "Integer", 1, val, 0;
  [res] := obj;
  ghost 'fold $ObjFields(obj + 3; ∅)';
  ghost 'fold $ObjFields(obj + 1; {(obj + 1, "int", val)})';
  ghost 'fold $FieldRefs(; ∅, ∅)';
  ghost 'fold $FieldRefs(; {(obj + 1, "int", val)}, ∅)';
  ghost 'fold $Object(obj; ?; "Integer")';
  ghost 'fold $References(; {(obj, "Integer", {(obj + 1, "int", val)})}, ∅)';
  ghost 'fold $References(; ∅, {(obj, "Integer", {(obj + 1, "int", val)})})'
}
proc Integer_valueOf(@string, res)
  pre : res ↦ _;
  post : ∃ o, n.res ↦ o ⋆ $References(; ∅, {(o, "Integer", {(o + 1, "int", n)})})
          ⋆ $Cast(n; ; @string);
{
  locals tmp, int;
  tmp := new0;
  call Integer_parseInt(@string, tmp);
  int := [tmp];
  call Integer_construct(int, tmp);
  [res] := [tmp];
  disposetmp
}
proc abstract Integer_toString(i, res)
  pre : res ↦ _;
  post : ∃ @s.res ↦ @s ⋆ $Cast(i; ; @s);
proc abstract Integer_parseInt(@s, res)
  pre : res ↦ _;
  post : ∃ n.$Cast(n; ; @s) ⋆ res ↦ n
```

Figure 4.16: Members of the Integer wrapper class

```
forall pure $Cast(_;;_).
lemma abstract strint_injectivity()
    ∀n, @m, @n.
    pre : $Cast(n;;@m) ⋆ $Cast(n;;@n);
    post : @m = @n;
lemma abstract strint_zero()
    ∀@z.
    pre : $Cast(0;;@z);
    post : @z = "0";
;
```

Figure 4.17: Abstract modelling of string-integer conversion

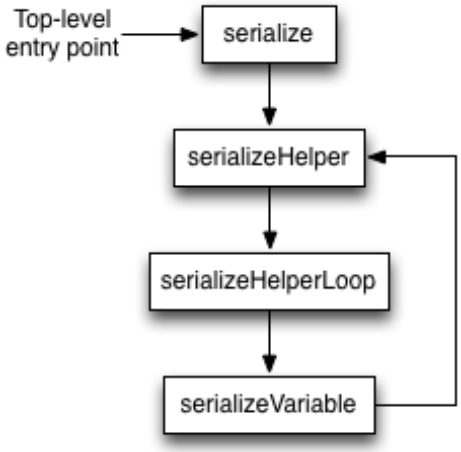


Figure 4.18: Procedure call tree showing the recursion of the serialization algorithm

associated and a small change in one will have an effect on the other procedures.

The individual procedures constituting the algorithm are now given. The code includes some annotations which guide the proof, in pale font, which can be ignored until the next section where the specifications are given and predicates defined.

serialize The top-level serialize method in Figure 4.19 accepts arguments for an object and a result pointer for storing the resulting XML structure. The body of the procedure begins by creating a new map for private use by the serialization algorithm, and an empty XML document. The entrance to the recursive algorithm is the procedure *serializeHelper*. After the serialization is complete, the map is de-allocated as it is only used internally.

serializeHelper The recursive algorithm for serialization is divided in to three procedures, the first of which, in Figure 4.20, creates a new XML record for a given object, before delegating the role of serializing the fields to the next procedure.

```
proc serialize(source, res) {
  locals doc, table;
  call IdentityHashMap_construct(res);
  table := [res];
  call XMLDocument_construct(res);
  doc := [res];
  ghost 'fold $MaybeCreatedObjXML(; ∅, ∅)';
  ghost 'fold $inMeta(; ∅, %M)'; ghost 'fold $MaxID(0; ∅)'
  call serializeHelper(source, doc, table);
  ghost 'lemma cast_zero()';
  call dispose_map(table);
  [res] := doc
}
```

Figure 4.19: Entry to the serialization algorithm

The body of the procedure first checks the current size of the map, and uses this as the identifier for the new object being added to the XML. Next, reflection is used to get the name of the class for the XML record, and then for retrieving the list of fields for the class. This list of fields is then handled by the procedure *serializeHelperLoop* which returns the complete XML record for the object and its fields. This is then added to the previously defined XML document.

serializeHelperLoop This procedure, in Figure 4.21 recursively processes each field of an object.

The body traverses the inductive list. In the empty (else) case no action is required. For the non-empty case, reflection is used to get the name of the current field, the declaring class name, and the class of the field. The current content of the field is then reflectively accessed through *Field_get*. The conversion of this field content to an XML value or reference element is delegated to the procedure *serializeVariable* which will handle the field different depending on whether it is a primitive integer or object reference. With the resulting value-as-XML, along with the earlier discovered field name and declaring class, a field XML record is created which is then added to the parent's (*target*'s) children. The procedure is then recursively called for the next field in the list.

serializeVariable The procedure in Figure 4.22 is used to handle each field contents, deciding whether it is a primitive integer or a complex object and creating an appropriate XML record containing either the integer value or the identifier of the object in the map.

The body of the code first uses reflection on the class of the given field to decide whether it is primitive. If it is, then the value of the field is converted to string and

```
proc serializeHelper(source, target, table) {
  locals tmp, size, fields, oElt, sourceclass, @id, @sourceclassName;
  tmp := new 0;
  call IdentityHashMap_size(table, tmp);
  size := [tmp];
  call Integer_toString(size, tmp);
  @id := [tmp]; [tmp] := 0;
  ghost 'lemma extract_obj_by_tuple()';
  call IdentityHashMap_put(table, source, @id);
  ghost 'lemma unfold_serializeDefined()';
  call Object_getClass(source, tmp);
  sourceclass := [tmp];
  call Class_getName(sourceclass, tmp);
  @sourceclassName := [tmp]; [tmp] := 0;
  call XMLObject_construct(@sourceclassName, @id, tmp);
  oElt := [tmp];
  call Class_getDeclaredFields(sourceclass, tmp);
  fields := [tmp];
  ghost 'fold $inMeta(; ∅, %M)';
  ghost 'fold $inMeta(; E, %M)';   // where E = {(oElt, @id, @sourceclassName, ∅)}
  ghost 'fold $MaybeCreatedField(; ∅, %doc ∪ %ascendants ∪ E)';
  ghost 'fold $MaybeCreatedObjXML(; ∅, %doc ∪ %ascendants ∪ E)';
  ghost 'fold $MaybeCreatedObjXML(; E, %doc ∪ %ascendants ∪ E)';
  ghost 'fold $References(; ∅, %refs)';
  ghost 'lemma size_notin_map()';
  ghost 'lemma maybeCreatedObjXML_subset()';
  ghost 'fold $MaxID(size + 1; %tbl ∪ {(source, @id)})';
  call serializeHelperLoop(fields, source, oElt, target, table);
  call XMLDocument_addContent(target, oElt);
  ghost 'lemma join_maybeCreatedObjXML()';
  ghost 'lemma join_inMeta()';
  call dispose_fieldList(fields);
  dispose tmp
}
```

Figure 4.20: Serialization procedure which takes an object, and adds it to the XML

```
proc serializeHelperLoop(fields, source, oElt, target, table) {
  locals child, declClass, fElt, fieldType, @name, next, serialized, tmp,
    @declClassStr, @declaringclass, current;
  ghost 'unfold $FieldList(fields; %fs)';
  if fields ≠ 0 then {
    tmp := new 0;
    current := [fields];
    call Field_getName(current, tmp);
    @name := [tmp];
    call Field_getDeclaringClass(current, tmp);
    declClass := [tmp];
    call Class_getName(declClass, tmp);
    @declClassStr := [tmp];
    ghost 'lemma split_fieldsExist_on_decl()';
    ghost 'lemma fields_in_metadata()';
    call Field_getType(current, tmp);
    fieldType := [tmp];
    ghost 'lemma extract_obj_by_tuple()';
    call Field_get(current, source, tmp);
    child := [tmp];
    ghost 'fold $References(; ∅, %refs)';
    ghost 'lemma unfold_fieldRefs()';
    ghost 'lemma refs_unique_ptrs()';
    ghost 'lemma meta_unique_class_name()';
    call serializeVariable(fieldType, child, target, table, tmp);
    serialized := [tmp];
    call XMLField_construct(@name, @declClassStr, serialized, tmp);
    fElt := [tmp];
    call XMLObject_addContent(oElt, fElt);
    next := [fields + 1];
    ghost 'lemma inMetaExtendedFields()';
    ghost 'lemma maybeCreatedObjXML_subset()'; ... × 2
    ghost 'lemma maybeCreatedObjXML_update_fields()';
    call serializeHelperLoop(next, source, oElt, target, table);
    dispose tmp
  } else { skip };
  ghost 'fold $FieldList(fields; %fs)'
}
```

Figure 4.21: Takes a list of fields, creates an XML record for each, and adds to parent object's XML record

```
proc serializeVariable(fieldType, child, target, table, res) {
  locals @childStr, size, tmp, @childInTbl, @sizeStr,
  @referenceStr, @valueStr;
  call Class_isPrimitive(fieldType, res); tmp := [res];
  if tmp ≠ 1 then {   // Field is an object
    @referenceStr := "reference";
    call IdentityHashMap_containsKey(table, child, res);
    tmp := [res];
    if tmp = 1 then {
      call IdentityHashMap_get(table, child, res);
      @childInTbl := [res]; [res] := 0;
      call XMLRefVal_construct(@referenceStr, @childInTbl, res)
    } else {
      call IdentityHashMap_size(table, res);
      size := [res];
      call Integer_toString(size, res);
      @sizeStr := [res]; [res] := 0;
      call XMLRefVal_construct(@referenceStr, @sizeStr, res);
      call serializeHelper(child, target, table)
      ghost 'lemma cast_injectivity()'
    }
  }else{   // Field is integer primitive
    @valueStr := "value";
    call Integer_toString(child, res);
    @childStr := [res]; [res] := 0;
    call XMLRefVal_construct(@valueStr, @childStr, res)
  }
}
```

Figure 4.22: Takes the content of a field and creates either a "reference" or "value" XML record. Recursively calls serializeHelper if content is an object not already serialized.

a "value" XML record created. If the field is not primitive, then the first stage is to see whether the referenced object has already been serialized, which can be deduced by possible membership in the map. If present, the corresponding identifier in the map is used to create a "reference" type XML record. If the referenced object is not already serialized, then the next available identifier (i.e. the map size) is used for creating the XML record, after which the serialization is recursively called on the referenced object. Note that the identifier is not added to the table here, because it is done in the body of the *serializeHelper*.

4.2.3.2 Specification and verification

The specifications that follow are complex, and it is recommended that they be inspected after reading the textual descriptions.

Before the specifications of the above procedures are presented, a number of predicates are defined which describe properties over combinations of the sets representing XML data, metadata, object closure, and the map used by the algorithm. These properties largely describe senses of well-definedness, such as the classes appearing in the XML must also be present in the metadata.

The predicate definitions used by the serialization stage are given in Figure 4.23. The first three describe properties of the XML tree, with one predicate for each level (object → field → field value). It is important to note that these predicates are not needed for the verification of the serialization algorithm if considered on its own. They are rather an output of the serialization process and assert that, for every field reference, the stored identifier is also present as an object in the XML. In other words, all referenced objects have themselves also been serialized. This fact is required for the verification of the deserialization, which will be presented in Section 4.2.3. The definitions of $MaybeCreatedObjXML and $MaybeCreatedField are standard inductive (pure) definitions, with uniqueness declared on the identifier elements of the tuples. The leaf $MaybeCreatedRefVal, however, asserts the crucial property that if a reference XML tag is encountered, then the content appears as an identifier (at second projection) in an object tag. The definitions all make use of a constant set $\%allObjs$, which is passed through as the last set-argument in each predicate. This represents the entire collection of XML objects and is the important set for asserting the required property.

Predicate $inMeta asserts that objects appearing in the XML are consistent with the classes contained in the metadata. In particular that, for an object in the XML, 1) its class is in the metadata and 2) the (names of) fields in the XML are a subset of the names of fields for that class in the metadata.

Predicate $Defined is used to show that the fields of objects appearing in a $References set are consistent with the metadata. That is 1) the object's class is included in the metadata, 2) the fourth projection of the field set in the metadata (declaring class) matches the object's class, and 3) all the fields declared in the metadata are included as fields in the object instance (with the $FieldsExist predicate defined in Chapter 3).

$$\textsf{recdef } \$\text{MaybeCreatedRefVal}(;\%\mathit{allObjs};@\mathit{xmlType},@\mathit{value}) :=$$
$$@\mathit{xmlType} = \text{“}\texttt{value}\text{”} \quad \vee \quad @\mathit{xmlType} = \text{“}\texttt{reference}\text{”} \star @\mathit{value} \in \mathrm{proj}_2(\%\mathit{objs});$$

$$\textsf{recdef } \$\text{MaybeCreatedField}(;\%\mathit{fields},\%\mathit{allObjs}) := \%\mathit{fields} = \emptyset$$
$$\vee \exists\, a, @\mathit{name}, @\mathit{decclass}, @\mathit{name}, @\mathit{type}, @\mathit{text}, \%\mathit{rest}.$$
$$\$\text{MaybeCreatedRefVal}(;\%\mathit{allObjs};@\mathit{type},@\mathit{text})$$
$$\star\, \$\text{MaybeCreatedField}(;\%\mathit{rest},\%\mathit{allObjs})$$
$$\star\, \%\mathit{fields} = \{(a, @\mathit{name}, @\mathit{decclass}, @\mathit{type}, @\mathit{text})\} \cup \%\mathit{rest} \star a \notin \mathrm{proj}_1(\%\mathit{rest});$$

$$\textsf{recdef } \$\text{MaybeCreatedObjXML}(;\%\mathit{objs},\%\mathit{allObjs}) := \%\mathit{objs} = \emptyset$$
$$\vee \exists\, a, @\mathit{id}, @\mathit{class}, \%\mathit{fs}, \%\mathit{rest}.$$
$$\$\text{MaybeCreatedField}(;\%\mathit{fs},\%\mathit{allObjs}) \star \$\text{MaybeCreatedObjXML}(;\%\mathit{rest},\%\mathit{allObjs})$$
$$\star\, \%\mathit{objs} = \{(a, @\mathit{id}, @\mathit{class}, \%\mathit{fs})\} \cup \%\mathit{rest} \star @\mathit{id} \notin \mathrm{proj}_2(\%\mathit{rest});$$

$$\textsf{recdef } \$\text{inMeta}(;\%\mathit{objs},\%M) := \%\mathit{objs} = \emptyset$$
$$\vee \exists\, \%\mathit{rest}, a, @\mathit{id}, @\mathit{class}, \%\mathit{fs}, c, \mathit{cls}, \%\mathit{fieldTypes}, \%\mathit{ms}.$$
$$(\mathit{cls}, @\mathit{class}, \%\mathit{fieldTypes}, \%\mathit{ms}, c) \in \%M \star \mathrm{proj}_2(\%\mathit{fs}) \subseteq \mathrm{proj}_2(\%\mathit{fieldTypes})$$
$$\star\, \$\text{inMeta}(;\%\mathit{rest},\%M)$$
$$\star\, \%\mathit{objs} = \{(a, @\mathit{id}, @\mathit{class}, \%\mathit{fs})\} \cup \%\mathit{rest} \star @\mathit{id} \notin \mathrm{proj}_2(\%\mathit{rest});$$

$$\textsf{recdef } \$\text{Defined}(;\%\mathit{refs},\%M) := \%\mathit{refs} = \emptyset$$
$$\vee \exists\, \mathit{cPtr}, \mathit{construct}, \%\mathit{cFs}, \%\mathit{cMs}, \mathit{ptr}, @\mathit{type}, \%\mathit{fs}, \%\mathit{refsRest}.$$
$$(\mathit{cPtr}, @\mathit{type}, \%\mathit{cFs}, \%\mathit{cMs}, \mathit{construct}) \in \%M \star \mathrm{proj}_4(\%\mathit{cFs}) \subseteq \{@\mathit{type}\}$$
$$\star\, \$\text{FieldsExist}(\mathit{ptr};\%\mathit{cFs},\%\mathit{fs}) \star \$\text{Defined}(;\%\mathit{refsRest},\%M)$$
$$\star\, \%\mathit{refs} = \{(\mathit{ptr}, @\mathit{type}, \%\mathit{fs})\} \cup \%\mathit{refsRest} \star \mathit{ptr} \notin \mathrm{proj}_1(\%\mathit{refsRest});$$

$$\textsf{recdef } \$\text{MaxID}(\mathit{size};\%M) := \%M = \emptyset$$
$$\vee \exists\, x, @y, \%\mathit{rest}, n. \quad \$\text{Cast}(n;;@y) \star n < \mathit{size}$$
$$\star\, \$\text{MaxID}(\mathit{size}-1;\%\mathit{rest}) \star \%M = \{(x, @y)\} \cup \%\mathit{rest} \star x \notin \mathrm{proj}_1(\%\mathit{rest});$$

Figure 4.23: Predicates used for specifying the serialization algorithm

proc $serialize(source, res)$
$\forall \%M, \%fs, @type, \%refs, intCls.$
pre : $\$\text{Meta}(;\%M) \star (source, @type, \%fs) \in \%refs \star \$\text{References}(;\emptyset, \%refs) \star res \mapsto \_$
$\star \$\text{Defined}(;\%refs, \%M) \star (intCls, \text{“int”}, \emptyset, \emptyset, 0) \in \%M;$
post : $\exists d, \%xml, \%xmlFs, oElt.$
$res \mapsto d \star \$\text{Meta}(;\%M) \star \$\text{References}(;\emptyset, \%refs) \star \$\text{XMLDocument}(d;\%xml)$
$\star \$\text{inMeta}(;\%xml, \%M) \star \$\text{MaybeCreatedObjXML}(;\%xml, \%xml)$
$\star (oElt, \text{“0”}, @type, \%xmlFs) \in \%xml;$

Figure 4.24: Specification of *serialize*

Finally, a predicate is defined which places a restriction on the map used by the serialization. The definition is intended to say that the current size of the map is not being used as an identifier for one of the elements. Recall that elements are added to the map using the size to be the next identifier. The definition traverses each element, uses $Cast to get the integer value of the string, and assert that that identifier is less than the size.

The specifications of the four serialization procedures are now given in Figures 4.24 to 4.27, and explained below. Additionally, notable annotations that were included in the code are also now explained.

serialize The specification requires the metadata, the object being serialized within a $References closure, a pointer for storing the result, the fact that the objects are well-defined with the metadata, and that there is a class in the metadata for the primitive int type. The post-condition leaves all the objects and metadata the same, and says that the result contains a pointer to the created XML document. Additionally, the post-condition ensures that the objects recorded in the XML are of classes in the metadata ($inMeta), that the XML is well-defined in that referenced objects for all fields are also in the XML ($MaybeCreatedObjXML), and that there is a root object, with an identifier 0 in the XML record, whose class matches that of the argument object.

Most of the annotations seen in the code in Figure 4.19 create initial “empty” instances of the inductive predicates that are required for invoking the *serializeHelper* procedure. The other hint is one of the lemmas created for the integer conversion handling, which allows one to show that the XML identifier created for the source object by *serializeHelper* for an initially empty table will be “0”. The lemma simply says $\$\text{Cast}(0;;@s) \Rightarrow @s = \text{“0”}$.

serializeHelper The specification of *serializeHelper* demonstrates an unusual requirement, namely that not all XML object records will have been added to the main document. This situation arises because the object record is not added to the document until it is complete, with all fields included, however the possible re-entrant call to *serializeHelper*

$$
\begin{array}{l}
\mathsf{proc}\ \mathit{serializeHelper}(\mathit{source}, \mathit{target}, \mathit{table}) \\
\forall\, \%M, \%\mathit{fs}, @\mathit{type}, \mathit{msize}, \%\mathit{tbl}, z, \%\mathit{doc}, \%\mathit{refs}, \%\mathit{ascendants}, \mathit{intCls}. \\
\quad \mathsf{pre} : \$\mathrm{Meta}(;\%M) \star \$\mathrm{XMLDocument}(\mathit{target};\%\mathit{doc}) \star \$\mathrm{IDMap}(\mathit{table}, \mathit{msize};\%\mathit{tbl}) \\
\qquad \star\, \$\mathrm{References}(;\emptyset,\%\mathit{refs}) \star (\mathit{source}, @\mathit{type}, \%\mathit{fs}) \in \%\mathit{refs} \star \mathit{source} \notin \mathrm{proj}_1(\%\mathit{tbl}) \\
\qquad \star\, \$\mathrm{inMeta}(;\%\mathit{doc},\%M) \star \$\mathrm{Defined}(;\%\mathit{refs},\%M) \\
\qquad \star\, \$\mathrm{MaybeCreatedObjXML}(;\%\mathit{doc},\%\mathit{doc} \cup \%\mathit{ascendants}) \star \$\mathrm{MaxID}(\mathit{msize};\%\mathit{tbl}) \\
\qquad \star\, (\mathit{intCls}, \text{“}\texttt{int}\text{”}, \emptyset, \emptyset, 0) \in \%M \star \mathrm{proj}_2(\%\mathit{doc} \cup \%\mathit{ascendants}) = \mathit{proj}(2, \%\mathit{tbl}) \\
\qquad \star\, \mathrm{proj}_2(\%\mathit{ascendants}) \cap \mathrm{proj}_2(\%\mathit{doc}) = \emptyset; \\
\quad \mathsf{post} : \exists\, \mathit{oElt}, @\mathit{id}, \%\mathit{newXmlFs}, \mathit{newmsize}, \%\mathit{newObjs}, \%\mathit{newTbl}. \\
\qquad \$\mathrm{Meta}(;\%M) \star \$\mathrm{XMLDocument}(\mathit{target};\%\mathit{newObjs}) \\
\qquad \star\, (\mathit{oElt}, @\mathit{id}, @\mathit{type}, \%\mathit{newXmlFs}) \in \%\mathit{newObjs} \star (\mathit{source}, @\mathit{id}) \in \%\mathit{newTbl} \\
\qquad \star\, \%\mathit{doc} \subseteq \%\mathit{newObjs} \star \%\mathit{tbl} \subseteq \%\mathit{newTbl} \star \mathit{msize} \subseteq \mathit{newmsize} \\
\qquad \star\, \$\mathrm{IDMap}(\mathit{table}, \mathit{newmsize};\%\mathit{newTbl}) \star \$\mathrm{References}(;\emptyset,\%\mathit{refs}) \star \$\mathrm{Cast}(\mathit{msize};;@\mathit{id}) \\
\qquad \star\, \$\mathrm{MaybeCreatedObjXML}(;\%\mathit{newObjs},\%\mathit{newObjs}\%\mathit{ascendants}) \\
\qquad \star\, \$\mathrm{MaxID}(\mathit{newmsize};\%\mathit{newTbl}) \star \mathrm{proj}_2(\%\mathit{newObjs} \cup \%\mathit{ascendants}) = \mathrm{proj}_2(\%\mathit{newTbl}) \\
\qquad \star\, \$\mathrm{inMeta}(;\%\mathit{newObjs},\%M) \star \mathrm{proj}_2(\%\mathit{ascendants}) \cap \mathrm{proj}_2(\%\mathit{newObjs}) = \emptyset;
\end{array}
$$

Figure 4.25: Specification of *serializeHelper*

needs to be aware of the “pending” object (or objects as the recursion gets deeper). For this reason, the set $\%\mathit{ascendants}$ appears which will contain all the XML objects in progress, but not yet in the XML document.

The pre-condition requires the metadata, the XML created thus far, the map, and the closure of objects for the given argument object *source*. Additionally, the pre-condition requires that the given object is not already in the map (i.e. has not already been seen and serialized), the four well-definedness properties from Figure 4.23, the primitive integer class's presence in the metadata, that those identifiers in the map are exactly those in the XML, and that identifiers from XML objects not yet added to the document do not overlap with those already present.

The post-condition reflects the new state of the XML and map, which will be supersets of the starting ones, including the fact that a record exists in the XML for the source object given in the arguments. Due to the mutual recursion the post-condition also includes the same properties from the pre-condition, updated for the new sets.

There are a number of lemmas used in guiding the proof search, with only four not performing standard splitting. The first, *size_notin_map* uses the $MaxID predicate to assert that the original size of the map did not appear in the map as an element key. The second, *maybeCreatedObjXML_subset*, allows the extension of the second argument of $MaybeCreatedObjXML providing that the second-projection of the new value is a superset of the old. The last two join two instances of predicates that use the set representation

proc $serializeHelperLoop(fields, source, oElt, target, table)$

$\forall\, @class, \%M, cPtr, \%cFs, \%cMs, construct, \%sourceFs, \%remainingSourceFs, \%objs,$
$\%xmlFs, msize, \%tbl, @id, \%fs, \%refs, \%obj, \%doc, \%ascendants, intCls.$

pre :

$\$\text{Meta}(;\%M) \star \$\text{FieldList}(fields;\%fs)$
$\star\, \$\text{XMLObject}(oElt;\%obj) \star \%obj = \{(oElt, @id, @class, \%xmlFs)\}$
$\star\, \$\text{XMLDocument}(target;\%doc) \star \$\text{IDMap}(table, msize;\%tbl)$
$\star\, (source, @class, \%sourceFs) \in \%refs \star \$\text{References}(;\emptyset, \%refs)$
$\star\, (cPtr, @class, \%cFs, \%cMs, construct) \in \%M$
$\star\, \$\text{FieldsExist}(source;\%fs, \%remainingSourceFs) \star \%remainingSourceFs \subseteq \%sourceFs$
$\star\, \%fs \subseteq \%cFs \star \text{proj}_2(\%obj \cup \%doc \cup \%ascendants) = proj(2, \%tbl)$
$\star\, proj(4, \%cFs) \subseteq \{@class\} \star \$\text{Defined}(;\%refs, \%M)$
$\star\, \$\text{MaybeCreatedObjXML}(;\%doc, \%doc \cup \%ascendants \cup \%obj)$
$\star\, \$\text{MaybeCreatedObjXML}(;\%obj, \%doc \cup \%ascendants \cup \%obj)$
$\star\, \text{proj}_2(\%doc) \cap \text{proj}_2(\%ascendants \cup \%obj) = \emptyset \star (intCls, \text{“}\texttt{int}\text{”}, \emptyset, \emptyset, 0) \in \%M$
$\star\, \$\text{MaxID}(msize;\%tbl) \star \$\text{inMeta}(;\%obj, \%M) \star \$\text{inMeta}(;\%doc, \%M);$

post : $\exists\, newmsize, \%newTbl, \%newXmlFs, \%newDoc, \%newObj.$

$\$\text{Meta}(;\%M) \star \$\text{FieldList}(fields;\%fs)$
$\star\, \$\text{XMLObject}(oElt;\%newObj) \star \%newObj = \{(oElt, @id, @class, \%newXmlFs)\}$
$\star\, \$\text{XMLDocument}(target;\%newDoc) \star \$\text{inMeta}(;\%newDoc, \%M)$
$\star\, \$\text{inMeta}(;\%newObj, \%M) \star \%doc \subseteq \%newDoc \star \%tbl \subseteq \%newTbl$
$\star\, msize \leq newmsize \star \text{proj}_2(\%newObj \cup \%newDoc \cup \%ascendants) = \text{proj}_2(\%newTbl)$
$\star\, \$\text{IDMap}(table, newmsize;\%newTbl) \star \$\text{References}(;\emptyset, \%refs)$
$\star\, \$\text{MaxID}(newmsize;\%newTbl)$
$\star\, \$\text{MaybeCreatedObjXML}(;\%newDoc, \%newDoc \cup \%newObj \cup \%ascendants)$
$\star\, \$\text{MaybeCreatedObjXML}(;\%newObj, \%newDoc \cup \%newObj \cup \%ascendants)$
$\star\, \text{proj}_2(\%newDoc) \cap proj(2, \%ascendants \cup \%newObj) = \emptyset;$

Figure 4.26: Entry to the serialization algorithm

of the XML structure, so long as the identifiers in each are disjoint.

serializeHelperLoop Recall that this procedure takes a list of fields, and builds a list of XML field records. Due to the recursive nature of the algorithm the specification must include all the parts from the previous procedure, which includes unfolded parts from the $Defined predicate. The pre-condition also now requires an $XMLObject instance, which is a singleton of the object currently being handled, to which the fields will be added. Additionally the list of fields contained in the metadata for the parent object's class ($\%cFs$) must be a superset of the list of fields yet to be processed ($\%fs$). The post-condition is

largely the same as with the previous procedure, representing the same properties for the new state after more objects and fields have been serialized.

The lemmas necessary for automated proof are mostly for unfolding inductive lists, with the exception of the following:

- *fields_in_metadata* : uses the $Defined predicate to reveal that a field of an object in a $References set must also appear in the metadata.
- *refs_unique_ptrs* : uses the separation facts in the $References predicate to assert that two objects in the set with the same pointer must refer to the same object.
- *meta_unique_class_name* : uses the uniqueness of the class name in the metadata to show that two elements with the same name must be identical.
- *inMeta_extended_fields* : show that the XML data can be updated and still be well-defined with respect to the metadata, so long as the newly added data is in the metadata.
- *maybeCreatedObjXML*$_s$*ubset* : show that the second argument to and instance of $MaybeCreatedObjXML, which contains all seen objects, can be safely modified so long as the second-projection of the new value is a superset of the old.
- *maybeCreatedObjXML_update_fields* : show that the XML data is still well-defined with respect to itself when an additional field is added if that fields content identifier is in the new XML.

serializeVariable Recall that this procedure accepts the contents of a field, and creates a reference or value record as appropriate, with different handling depending on whether the content is a primitive type or not. Accordingly, the specification is split in to two cases. The second case is the simplest, requiring only the metadata, and the fact that the int class exists in the metadata. The corresponding post-condition ensures an XML record has been created with the type attribute set to "value" and the content containing the string cast of the integer value of the *child* argument.

The more complex case handles objects. The pre-condition requires the metadata, the object/identifier map, the closure of the object contained in the field, the current XML data, the four well-definedness properties, that all the objects in the XML (including unfinished "ascendants") appear in the map, and that there is no overlap with identifiers. Much of this is not used directly by the *serializeVariable* procedure, but in a recursive call to the above *serializeHelper*. The post-condition ensures that the result is an XML record of "reference" type, containing an object who's identifier appears in the XML data. The XML and map may have been expanded, due to the serialization of the fields of the object given in the argument.

There is one lemma which is necessary because of the lack of built-in conversion between strings and integers. The lemma simply states that converting from the same integer will result in the same string each time.

proc $serializeVariable(fieldType, child, target, table, res)$

$\forall\, \%M, \%tbl, msize, \%doc, \%refs, \%childFs, @class, \%cFs, \%cMs, construct,$
$\%ascendants, \%fieldRefs, \%xmlFs, intCls.$

pre : $@class \neq$ "`int`"
$\star\ \$\mathrm{Meta}(;\%M) \star \$\mathrm{IDMap}(table, msize; \%tbl) \star res \mapsto \_ \star \$\mathrm{References}(;\emptyset, \%refs)$
$\star\ (child, @class, \%childFs) \in \%refs \star \$\mathrm{XMLDocument}(target; \%doc)$
$\star\ (fieldType, @class, \%cFs, \%cMs, construct) \in \%M \star \$\mathrm{inMeta}(;\%doc, \%M)$
$\star\ \$\mathrm{Defined}(;\%refs, \%M) \star \mathrm{proj}_2(\%doc \cup \%ascendants) = \mathrm{proj}_2(\%tbl)$
$\star\ \$\mathrm{MaybeCreatedObjXML}(;\%doc, \%doc \cup \%ascendants)$
$\star\ \mathrm{proj}_2(\%ascendants) \cap \mathrm{proj}_2(\%doc) = \emptyset$
$\star\ \$\mathrm{MaxID}(msize; \%tbl) \star (intCls, $"`int`"$, \emptyset, \emptyset, 0) \in \%M$
$\vee$
$@class =$ "`int`" $\star\ \$\mathrm{Meta}(;\%M) \star (fieldType, @class, \emptyset, \emptyset, 0) \in \%M \star res \mapsto \_$
;

post : $\exists\, rv, @value, \%newDoc, \%newTbl, newmsize, @xmlType.$
$@class \neq$ "`int`" $\star\ @xmlType =$ "`reference`"
$\star\ \$\mathrm{Meta}(;\%M) \star \$\mathrm{IDMap}(table, newmsize; \%newTbl) \star res \mapsto rv$
$\star\ \$\mathrm{XMLRefVal}(rv;;@xmlType, @value) \star \$\mathrm{XMLDocument}(target; \%newDoc)$
$\star\ \$\mathrm{References}(;\emptyset, \%refs) \star @value \in \mathrm{proj}_2(\%newDoc \cup \%ascendants)$
$\star\ \mathrm{proj}_2(\%newDoc \cup \%ascendants) = \mathrm{proj}_2(\%newTbl)$
$\star\ \mathrm{proj}_2(\%ascendants) \cap \mathrm{proj}_2(\%newDoc) = \emptyset \star \%doc \subseteq \%newDoc \star \%tbl \subseteq \%newTbl$
$\star\ msize \leq newmsize \star \$\mathrm{MaxID}(newmsize; \%newTbl) \star \$\mathrm{inMeta}(;\%newDoc, \%M)$
$\star\ \$\mathrm{MaybeCreatedObjXML}(;\%newDoc, \%newDoc \cup \%ascendants)$
$\vee$
$\exists\, rv, @xmlType, @value.$
$@class =$ "`int`" $\star\ @xmlType =$ "`value`" $\star\ \$\mathrm{Meta}(;\%M) \star res \mapsto rv$
$\star\ \$\mathrm{XMLRefVal}(rv;;@xmlType, @value) \star \$\mathrm{Cast}(child;;@value);$

Figure 4.27: Entry to the serialization algorithm

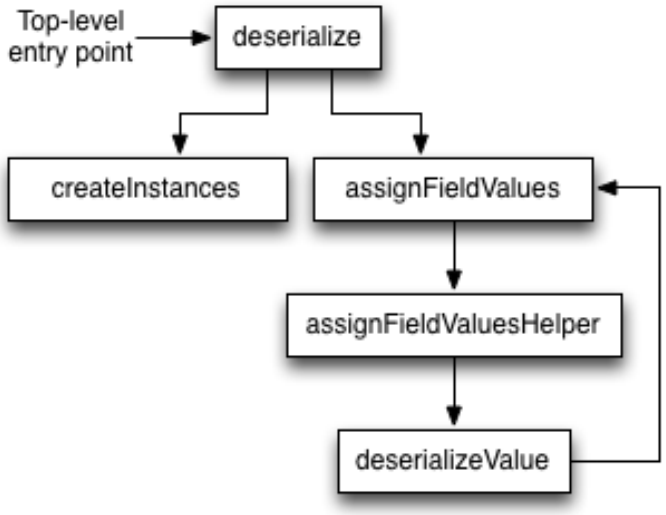


Figure 4.28: Procedure call tree showing the recursion of the deserialization algorithm

4.2.4 Deserialization

4.2.4.1 The program

The process of deserialization has two main stages. First, a new object instance is created for every object in the XML data. Next, for each object in the XML the fields are appropriately connected, or new Integer objects created for fields that were primitive. The deserialization requires the use of another map for keeping track of created objects based on the identifier in the XML. This is the opposite of the map used for the serialization. In Figure 4.28 the call-tree of the deserialization algorithm is given, where two of the procedures are recursive. The implementation of these procedures is now given. As with the serialization, the annotations in pale font can be ignored until the next section.

deserialize The top-level deserialization procedure accepts an XML document, and returns the closure of the root object. The body simply creates a map for use by the deserialization algorithm, then calls the procedure which creates all the objects found in the XML. The *assignFieldValues* procedure then connects all the objects as described by the field information in the XML. Finally the pointer to the root object (with identifier 0) is retrieved from the table and returned.

createInstances The purpose of this procedure is to create new instances of all the objects described in the XML data. Initially, all the fields are fresh and hence the objects from the XML will all be unconnected. The procedure traverses the XML recursively and begins by retrieving a handle to the class in the metadata relating to the current object record. The constructor is then retrieved, and invoked reflectively to create the new fresh object instance. A pointer to this object is then stored in the map, indexed by the corresponding identifier used in the XML record. Recall that the example has

```
proc deserialize(source, res) {
  locals table;
  call HashMap_construct(res);
  table := [res];
  ghost 'fold $References(; ∅, ∅)';
  call createInstances(table, source);
  call assignFieldValues(table, source);
  call HashMap_get(table, "0", res);
  ghost 'lemma unfold_ci2()';
  ghost 'lemma refs_unique_ptrs()';
  call dispose_table(table)
}
```

been simplified, thanks to the specific structure of the XML, and retrieval of attributes (@*clsName*, @*oID*) can be done with direct dereferencing.

assignFieldValues Once all the objects are created, the field pointers can be appropriately arranged by this procedure. It recursively processes each object, delegating the actual field updating to a helper procedure. The body traverses the list of object XML records, first retrieving the unique identifier and using it to get a pointer to that object from the map. The helper procedure is then used, whose task is to process the list of fields.

assignFieldValuesHelper This procedure iterates through the children of an object in the XML, each of which represents a field. In terms of the original Java algorithm [50], the procedure replaces the while-loop in the *assignFieldValues* method. For each field, the value of the respective field in the newly created object is updated with either the pointer to the referenced object, or a new Integer object if the field was primitive.

The non-empty case of the procedure body retrieves the declaring-class name record of the field being examined in the XML data, and uses reflection to look up the class by string name (*Class_forName*). This gives a handle to the class in the metadata which is then used to look up the field (by name), based on the name attribute in the XML. Next the *deserializeValue* procedure creates and Integer object or returns a pointer to an already created object, which is then used to update the field value using the reflective procedure *Field_set*. The procedure then proceeds recursively.

deserializeValue This procedure processes an individual field in the XML, and either returns a pointer to an existing object if the field contained an identifier to another object, or returns a new "Integer" object if the field is primitive. The body is fairly simple, first checking the type attribute of the field XML data: if "reference", then it simply looks up the identifier in the table; otherwise, it create a new Integer object. The procedure

```
proc createInstances(table, objList) {
  locals tmp, next, @clsName, cls, cons, instance, @oID;
  ghost 'unfold $XMLObject(objList; %L)';
  if objList ≠ 0 then {
    tmp := new 0;
    @clsName := [objList];
    ghost 'lemma unfold_inMeta()';
    call Class_forName(@clsName, tmp);
    cls := [tmp];
    call Class_getConstructor(cls, tmp);
    cons := [tmp];
    call Constructor_newInstance(cons, tmp);
    ghost 'lemma join_references()';
    instance := [tmp];
    @oID := [objList + 1];
    call HashMap_put(table, @oID, instance);
    next := [objList + 3];
    call createInstances(table, next);
    ghost 'fold $CI(; %L, %M, ?, ?)';
    dispose tmp
  } else {
    ghost 'fold $CI(; %L, %M, %tbl, %refs)';
    skip
  };
  ghost 'fold $XMLObject(objList; %L)'
}
```

Integer_valueOf is the special Integer constructor from Section 4.2.2, which additionally converts from a string.

4.2.4.2 Specification and verification

As was mentioned during the specification of the algorithm for serialization, the "inMeta" and "MaybeCreated" predicates can be considered output from the serialization process and then appear in the specifications that follow and are important for the verification. Additionally, there is one more predicate that is relevant to the deserialization which describes the state after the *createInstances* procedure has run. This predicate is given in Figure 4.29 and is defined inductively on the XML structure. For a particular object in the XML, it ensure that 1) an object has been created on the heap, associated with the relevant identifier by the map, 2) that the created object's fields are consistent with those

```
proc assignFieldValues(table, objList) {
  locals tmp, @oID, instance, fElts, next;
  tmp := new 0;
  ghost 'unfold $XMLObject(objList; %objs)';
  if objList ≠ 0then {
    @oID := [objList + 1];
    call HashMap_get(table, @oID, tmp);
    instance := [tmp];
    fElts := [objList + 2];
    ghost 'lemma unfold_maybeCreatedObjXML()';
    ghost 'lemma unfold_ci()';
    call assignFieldValuesHelper(table, fElts, instance);
    next := [objList + 3];
    call assignFieldValues(table, next)
  } else {
    skip
  };
  ghost 'fold $XMLObject(objList; %objs)';
  dispose tmp
}
```

in the metadata (by $FieldsExist), and 3) that all of the fields mentioned in the XML also appear in the metadata.

The specifications for the five deserialization procedures are now given.

deserialize The specification of the top level deserialization procedure in Figure 4.30 requires the XML data, metadata, the $inMeta predicate describing well-definedness of the XML with metadata, well-definedness of the XML with respect to referenced objects also being in the XML, and the guarantee of a root object element in the XML. The post-condition states that a set of objects have been created, wrapped up by a $References predicate instance, and it includes at least one object which is of the same type as the XML root object described in the pre-condition. The result pointer then points to that object. The XML document remains unchanged.

The two lemmas, *unfold_ci2* and *refs_unique_ptrs*, first unfold an instance of the $CI predicate and then show that two objects with the same pointer in a $References set must be the same object.

createInstances Also in Figure 4.30, the pre-condition requires the the current map, the XML, the metadata, the current collection of objects, the fact that those objects in

```
proc assignFieldValuesHelper(table, fieldList, instance) {
  locals @className, f, fieldDC, @fieldName, tmp, vElt, value, next;
  tmp := new 0;
  ghost 'unfold $XMLField(fieldList; %fields)';
  if fieldList ≠ 0 then {
    @className := [fieldList + 1];
    call Class_forName(@className, tmp);
    fieldDC := [tmp];
    @fieldName := [fieldList]; [tmp] := 0;
    call Class_getDeclaredField(fieldDC, @fieldName, tmp);
    f := [tmp];
    vElt := [fieldList + 2];
    ghost 'lemma unfold_maybeCreatedFields()';
    call deserializeValue(vElt, table, tmp);
    value := [tmp];
    ghost 'lemma split_fieldsExist_on_decl()';
    ghost 'lemma extract_obj_by_tuple()';
    call Field_set(f, instance, value);
    ghost 'lemma update_external_fields()';
    ghost 'fold $References(; ∅, {(instance, @type, %updatedFs?)} ∪ %newRefs?)';
    next := [fieldList + 3];
    ghost 'lemma fields_exist_ignores_value()';
    call assignFieldValuesHelper(table, next, instance)
  } else {
    skip
  };
  ghost 'fold $XMLField(fieldList; %fields)';
  dispose tmp
}
```

```
proc deserializeValue(vElt, table, res) {
  locals @valtype, @vEltText;
  call XMLRefVal_getName(vElt, res);
  @valtype := [res];
  call XMLRefVal_getText(vElt, res);
  @vEltText := [res];
  if @valtype = "reference" then {   // Object field
    ghost 'unfold $MaybeCreatedRefVal(; ?; @name, @text)';
    call HashMap_get(table, @vEltText, res);
  } else {   // Integer field
    call Integer_valueOf(@vEltText, tmp);
    ghost 'lemma join_references()';
  }
}
```

$$
\begin{aligned}
&\textsf{recdef } \$\mathrm{CI}(;\%objs, \%M, \%tbl, \%refs) := \%objs = \emptyset \\
&\vee \exists\, \%rest, a, @id, @class, \%fs, cPtr, construct, \%cFs, \%cMs, \%objFs, oPtr. \\
&\quad \mathrm{proj}_2(\%fs) \subseteq \mathrm{proj}_2(\%cFs) \star (@id, oPtr) \in \%tbl \star (oPtr, @class, \%objFs) \in \%refs \\
&\quad \star\, (cPtr, @class, \%cFs, \%cMs, construct) \in \%M \star \$\mathrm{FieldsExist}(oPtr; \%cFs, \%objFs) \\
&\quad \star\, \$\mathrm{CI}(;\%rest, \%M, \%tbl, \%refs) \\
&\quad \star\, \%objs = \{(a, @id, @class, \%fs)\} \cup \%rest \star @id \notin \mathrm{proj}_2(\%rest);
\end{aligned}
$$

Figure 4.29: Additional predicate used for specifying the deserialization algorithm

the XML data have not yet been added to the map, the guarantee that all objects which have been processed (ie are in the map) are also in the current collection of objects, and the XML/metadata well-definedness. The post-condition describes the state where the map and object collection have been extended, maintaining the same properties as the pre-condition, along with the state matching that described by the $CI predicate.

There was one non-splitting lemma that appeared in the sourcecode annotation, making use of the $References definition. The lemma *join_references* is used to join two instances of $References, providing the object pointer part of the "externals" sets are disjoint, which is assured at this point because the sets are empty in both cases.

assignFieldValues The specifications for the procedures tasked with appropriately connecting the created objects are in Figure 4.31. Beginning with *assignFieldValues*, the pre-condition requires the metadata, the XML data, the map, the new objects on the heap, the fact that the XML is well-defined, the fact that all objects referenced by the map are

proc $deserialize(source, res)$
$\forall \%elts, \%M, a, @class, \%fs, \%rest.$
pre : $\$\text{XMLDocument}(source; \%elts) \star \$\text{Meta}(; \%M) \star res \mapsto \_$
$\star \$\text{inMeta}(; \%elts, \%M) \star \$\text{MaybeCreatedObjXML}(; \%elts, \%elts)$
$\star (a, \text{“0”}, @class, \%fs) \in \%elts \star @class \neq \text{“}\texttt{int}\text{”};$
post : $\exists o, \%oFs, \%objs.$
$\$\text{XMLDocument}(source; \%elts) \star \$\text{Meta}(; \%M)$
$\star \$\text{References}(; \emptyset, \%objs) \star (o, @class, \%oFs) \in \%objs \star res \mapsto o;$

proc $createInstances(table, objList)$
$\forall \%tbl, \%L, \%M, \%refs.$
pre : $\$\text{HashMap}(table; \%tbl) \star \$\text{XMLObject}(objList; \%L) \star \$\text{Meta}(; \%M)$
$\star \$\text{References}(; \emptyset, \%refs) \star \text{proj}_2(\%L) \cap \text{proj}_1(\%tbl) = \emptyset \star \text{proj}_2(\%tbl) \subseteq \text{proj}_1(\%refs)$
$\star \$\text{inMeta}(; \%L, \%M);$
post : $\exists \%newTbl, \%newRefs.$
$\$\text{HashMap}(table; \%newTbl) \star \$\text{XMLObject}(objList; \%L) \star \$\text{Meta}(; \%M)$
$\star \$\text{References}(; \emptyset, \%newRefs) \star \%tbl \subseteq \%newTbl \star \%refs \subseteq \%newRefs$
$\star \$\text{CI}(; \%L, \%M, \%newTbl, \%newRefs) \star \text{proj}_2(\%newTbl) \subseteq \text{proj}_1(\%newRefs)$
$\star \text{proj}_2(\%L) \subseteq \text{proj}_1(\%newTbl);$

Figure 4.30: Specifications for deserialization procedures

on the heap, the fact that all the objects in the XML are in the map, and the predicate $CI reflecting the state after the objects have been initially created by *createInstances*.

In addition to the predicate unfolding lemmas, there is one other. Using the fact that the definition only requires set membership, the lemma *ci_ref_extension* allows the extension (to a superset) of the references set argument of a $CI instance.

assignFieldValuesHelper The specification is large, but describes properties of the XML and objects already seen previously. The pre-condition requires the XML data for the set of fields being processed, metadata, the deserialization map, the current arrangement of objects on the heap including the current object being handled, the fact that the XML is well-defined with itself, the fact that the names of the fields in the XML correspond to names of fields in the metadata, the fact that all of the “declaring class” elements in the field XML are of the same classtype as the object being handled, the fact that the object's class is in the metadata, that the object is well-defined with respect to the fields in the metadata, that all the objects in the XML appear in the map, and finally that the objects in the map appear on the heap.

The post-condition states that the XML, metadata, and map have been maintained, and that there is a superset of objects on the heap where the new fields of the object being

proc $assignFieldValues(table, objList)$

$\forall \%M, \%tbl, \%objs, \%refs, \%allObjs.$

pre : $\$\text{Meta}(;\%M) \star \$\text{XMLObject}(objList;\%objs) \star \$\text{HashMap}(table;\%tbl)$
$\star \$\text{References}(;\emptyset,\%refs) \star \$\text{CI}(;\%objs,\%M,\%tbl,\%refs)$
$\star \$\text{MaybeCreatedObjXML}(;\%objs,\%allObjs) \star \text{proj}_2(\%tbl) \subseteq \text{proj}_1(\%refs)$
$\star \text{proj}_2(\%allObjs) \subseteq \text{proj}_1(\%tbl) \star \%objs \subseteq \%allObjs;$

post : $\exists \%newRefs.$
$\$\text{Meta}(;\%M) \star \$\text{XMLObject}(objList;\%objs) \star \$\text{HashMap}(table;\%tbl)$
$\star \$\text{References}(;\emptyset,\%newRefs) \star \text{proj}_1(\%refs) \subseteq \text{proj}_1(\%newRefs);$

proc $assignFieldValuesHelper(table, fieldList, instance)$

$\forall \%fields, \%M, @type, \%objFs, \%refs, \%tbl, construct, fClass, \%fTypes, \%ms, \%objs.$

pre : $\$\text{XMLField}(fieldList;\%fields) \star \$\text{Meta}(;\%M) \star \$\text{HashMap}(table;\%tbl)$
$\star \$\text{References}(;\emptyset,\%refs) \star (instance, @type, \%objFs) \in \%refs$
$\star \$\text{MaybeCreatedField}(;\%fields,\%objs) \star \text{proj}_2(\%fields) \subseteq \text{proj}_2(\%fTypes)$
$\star \text{proj}_3(\%fields) \subseteq \{@type\} \star (fClass, @type, \%fTypes, \%ms, construct) \in \%M$
$\star \$\text{FieldsExist}(instance;\%fTypes,\%objFs) \star \text{proj}_2(\%objs) \subseteq \text{proj}_1(\%tbl)$
$\star \text{proj}_2(\%tbl) \subseteq \text{proj}_1(\%refs) \cup \{instance\};$

post : $\exists \%newRefs, \%updatedFs.$
$\$\text{XMLField}(fieldList;\%fields) \star \$\text{Meta}(;\%M) \star \$\text{HashMap}(table;\%tbl)$
$\star \$\text{References}(;\emptyset,\%newRefs) \star (instance, @type, \%updatedFs) \in \%newRefs$
$\star \text{proj}_1(\%refs) \subseteq \text{proj}_1(\%newRefs);$

proc $deserializeValue(vElt, table, res)$

$\forall @name, @text, \%tbl, \%refs, \%objs.$

pre : $\$\text{XMLRefVal}(vElt;;@name, @text) \star \$\text{HashMap}(table;\%tbl) \star res \mapsto \_$
$\star \$\text{References}(;\emptyset,\%refs) \star \$\text{MaybeCreatedRefVal}(;\%objs;@name, @text)$
$\star \text{proj}_2(\%objs) \subseteq \text{proj}_1(\%tbl) \star \text{proj}_2(\%tbl) \subseteq \text{proj}_1(\%refs);$

post : $\exists o, \%newRefs.$
$\$\text{XMLRefVal}(vElt;;@name, @text) \star \$\text{HashMap}(table;\%tbl) \star res \mapsto o$
$\star \$\text{References}(;\emptyset,\%newRefs) \star o \in \text{proj}_1(\%newRefs) \star \%refs \subseteq \%newRefs;$

Figure 4.31: Specifications for deserialization procedures which handle field values

handled may have grown (and hence the references set expanded) if a primitive field was processed, which is required to be handled as a new proper object of the Integer wrapper class.

Looking at the lemmas used for guiding the verification, in addition to the predicate splitting lemmas there are three others that expose facts that are hidden inside predicate definitions. Lemma *update_external_fields* is fairly complex, but essentially shows that it is safe to update the value of an object's field in a $References set, so long as the new value is also in the set such that the closure is preserved. Lemma *fields_exist_ignores_value* allows the updating of the object-fields argument of a $FieldsExist predicate, with the fact that the third part of the tuple is not used.

deserializeValue This procedure returns a pointer to an object which will either already exist, through a reference in the map, or be a new Integer object to represent a primitive. The pre-condition requires the XML `<reference>` or `<value>` record, the map, a result pointer, all the current objects on the heap, the fact that the objects in the XML appear in the map, and the fact that the objects referenced in the map are on the heap. The post-condition maintains the XML and map, and asserts that the contents of the result cell yields an object in the (possibly extended) set of objects. The lemma used here has been used earlier in *createInstances*, and merges the $References instance in the pre-condition with the new version that has been created by the Integer constructor.

4.2.5 Using the program

Whilst the focus of the case study has been verification of the serialization and deserialization algorithms, which make direct use of reflection, a brief demonstration of their utilization can be seen in Figure 4.32.

For some class "Test", a stronger specification than usual has been provided for the constructor. The additional formulae are the metadata, with inclusion of the "Test" class with an explicit set of fields, and the instance of the $Defined predicate in the post-condition. Recalling the definition of this predicate, it can be easily folded with the constraints on the metadata in the pre-condition. The $References and $FieldsExist predicates usage should be familiar from the pattern of constructors discussed in Chapter 3.

Owing to the added instance of the $Defined predicate, which is required in the pre-condition of *serialize*, the main procedure can proceed in the natural manner. The body simply creates a new object of class Test, serializes it, deserializes the resulting XML, and stores the original and deserialized objects in the two result pointers. Note that the pre-condition must include $meta \mapsto \_$ to trigger the metadata generation. The post-condition includes two instances of $References: first, the original object is the sole member with address x, but in the second case the set of objects has a weaker description. Due to the single field owned by objects of the Test class having the primitive integer type, the deserialized set of objects will actually contain two objects (the target object and a new object of the Integer wrapper class). However, due to the weakness of the specification for

```
proc abstract Test_construct(res)
  ∀ x, %M, cPtr, %cMs, %cFs, cons.
  pre : res ↦ _ ⋆ $Meta(; %M)
    ⋆ (cPtr, "Test", %cFs, %cMs, cons) ∈ %M ⋆ %cFs = {(x, "x", "int", "Test", 2)};
  post : ∃ t, %refs.
    res ↦ t ⋆ $References(; ∅, %refs) ⋆ {(t, "Test", {(t + 2, "int", 0)})} = %refs
    ⋆ $FieldsExist(t; %cFs, {(t + 2, "int", 0)}) ⋆ $Defined(; %refs, %M) ⋆ $Meta(; %M);

proc main(res)
  pre : res ↦ _, _ ⋆ meta ↦ _;
  post : ∃ x, y, %xfs, %yfs, %newRefs, %M.
    res ↦ x, y ⋆ $References(; ∅, {(x, "Test", %xfs)}) ⋆ $References(; ∅, %newRefs)
    ⋆ (y, "Test", %yfs) ∈ %newRefs ⋆ $Meta(; %M);
{
  locals o, xml;
  call Test_construct(res);
  o := [res];
  call serialize(o, res);
  xml := [res];
  call deserialize(xml, res);
  [res + 1] := [res];
  [res] := o;
  call XMLDocument_dispose(xml)
}
```

Figure 4.32: Using the serialization and deserialization algorithms in a program

deserialize, an assertion can only be made over the "top" object, of which the class type has been guaranteed to match the original.

4.3 Discussion

4.3.1 General

This chapter has shown how the reflective library and verification system described in the previous chapters can be applied to real program examples, with varying issues being highlighted in each case. Additionally, the two examples were suitably complimentary to test the library in that they each focussed on different aspects of the reflection, with the first use case reflecting on methods and the second requiring the support for reflection on fields.

Overall, the specifications for the library were shown to be strong enough to assure the desired behavioural properties of the procedures, for instance the elimination of the possibilities of the exceptions that occur in the Java language. The additional burden of carrying around the metadata in the specifications was not a major concern, as it is contained within a predicate that need never be unfolded. However the pure facts asserting the existence of elements in the metadata does need extra consideration, and can swell the specifications slightly. Whilst the specifications for the reflective library are generally small, it has become clear that specifications for programs using the library must become more complicated to ensure that there is the well-definedness/existence guarantees when performing reflecting operations on an object or metaobjects. The amount of extra formula needed in specifications is dependent on whether or not the reflection needs to use "nested" elements of the metadata, like fields or methods, as this will obviously require the parent class to be in the metadata to give a reference to the nested element set, as well as the actual field or method element. This will often mean that a class tuple is shown to be an element of the metadata set where several variables in the tuple may never be used (e.g. method and constructor variables, when only the fields are used).

An issue that occurred more in the use of the verification system for the verification of reflection programs than in previous examples was the need to use lemma procedures. Whilst often these were for standard "unfolding" of pure lists, there were a number of cases where the symbolic state during verification becomes too large for the SMT-solver to decide necessary entailments. The lemmas helped by simplifying the problems to their smallest size or by engineering syntactically identical assertions that are trivially identifiable by the SMT-solver.

Other cases where lemmas were commonly used was for cases where facts are hidden inside predicate instances, but need to become explicit. Lemmas are useful for this because they can extract only the pertinent formulae, keeping the symbolic state under control and allowing possibly more complex sequences of unfolds/folds to take place outside the scope of the program.

As mentioned in Chapter 3, there was a thought that object constructors are laborious to verify due to the necessary predicate foldings, although the code is only a couple of lines. This was shown to certainly be true after undertaking these examples, where each object constructor follows the same sequence of repetitive folds, varying by the number of fields the object contains. There would certainly appear to be scope here for future work to make this fully automated, and allow implementations of the constructors to be omitted.

The verification times of these examples can be fairly long, in part due to their size (lines-of-code), but also due to the complexity of the assertions given to the SMT-solver. The focus of this research has not been looking at efficiency of deciding pure entailments, and it is also quite likely that the efficiency can be improved with the current solvers by configuring them in the right way for the example, or by knowing details of their implementation to best take advantage of their strengths.

$$\begin{aligned}&\textsf{recdef}\ \$\mathrm{Post}(target, arg; \%targFsPre, \%M, \%T; @targetClass, @argClass) :=\\ &\quad \exists \%tFsPost, \%rPost.\\ &\qquad \$\mathrm{Object}(target; \%tFsPost; @targetClass)\\ &\qquad \star\ \$\mathrm{Tree}(arg; \%rPost; @argClass) \star \$\mathrm{Meta}(; \%M)\\ &\qquad \star\ \$\mathrm{VisitorFun}(; \%T, \%targFsPre, \%tFsPost) \quad // \text{ Previously \$Fun}\\ &\qquad \star \$\mathrm{TreeFun}(; \%T, \%rPost, \ldots);\end{aligned}$$

Figure 4.33: Altering the predicate definitions to allow the visitor to manipulate the data-structure (changes shaded)

A final general detail is that the reflective library described in Chapter 3 contains only those reflective features required by these examples. Whilst it covers a good range of the features in Java's reflection API, it is by no means complete. Further extensions to this work could be to include more such features, like accessibility modifiers or full constructor support.

4.3.2 Visitor

In terms of analysis of the individual examples and the way the verification model used here works, the visitor pattern is first discussed. A key point to be addressed, which embodies the usefulness of the visitor as a design pattern, is that the core aspect must be reusable in other settings, for a different concrete visitor and/or a different data structure. In this case, that means ensuring that the specifications are suitably reusable.

Firstly, if one is using a visitor that performs a different function, such as counting *all* nodes (of any type), the model can be left largely intact except for providing an alternate definition for the $Fun predicate (and possibly its dependent $NumLeaves). It would be trivial to extend the arity of the predicate to also be parametrized by the class of the visitor such that the definition can be a disjunction and describe the function of multiple visitors so that the $Pre/$Post predicate definitions remain unaltered. A second case of reuse could be where the visitor actually alters the contents of the data-structure, rather than having a read-only effect. This would not be possible using the current specification pattern, however the case can be easily accommodated by introducing an existential variable to describe the new structure in the $Post definition, and use another predicate to describe the relation between the new and old states. A sketch of this can be seen in Figure 4.33, where the occurrence of $Tree uses the new existential variable and the relation can be described by $TreeFun, which may additionally include other arguments as required that the behaviour is dependent on.

The third case of reuse to consider would be for a different data-structure. To handle this, the change will be largely limited to altering the $Tree predicate definition to describe an alternate structure (and possibly updating any relation predicate such as $TreeFun in

Figure 4.33). It is however worth mentioning that the example used in this thesis represents a non-cyclic binary tree. This has the advantage, with respect to the process of visiting, that the specification only contains the node objects of the subtree *remaining to visit*. If the tree were to contain cycles or multi-referenceable nodes, then earlier nodes would also need to be passed through to the recursive visit methods. Practically, this would probably involve including the entire data-structure in the specifications. To specify such structures, it would be fairly easy to create a specialized version of the $References predicate and use this instead. Indeed one can also envisage a setup where the original $References is used to describe the objects on the heap, which will give the desired closure for a recursive structure, and include an additional pure predicate to describe more concrete properties of the structure.

4.3.3 Serialization

With regards to the serialization/deserialization example, the first thing to note is the number of predicates required to describe the behaviour of the algorithm. The ability to provide user-defined predicates has been vital to making additional assertions over inductive data structures, like the XML structure. In most cases, however, it would be possible to merge the additional properties defined by some "pure" predicate directly into the predicate describing the structure. In cases where this is possible, it can be left up to the user whether or not they prefer "smaller" specifications in terms of number of predicates, or the separation of the core data structure definitions from example-specific specifications.

Secondly, something that was initially overlooked but became clear as a result of undertaking this example is when constructing an object using reflection, all the fields are given initial values of "blank" objects. If the fields are updated with new values, the original blank objects will remain in the heap because the *Field_set* procedure does not dispose of unused objects. To do so would necessitate checking whether they are referenced elsewhere. However, as this work is largely following the Java manner, this will be all taken care of by the garbage collection. The reason this is notable is because it has the disadvantage of upsetting the modularity principle of separation logic where specifications only need to contain the heaplets that will be required by the procedure. In this case, the $References will also contain the objects that will no longer be used and so the specifications will be larger than necessary. Recall that the definition of $References makes use of $\subseteq$, which makes it weaker than describing a true closure because it only ensures that the sets contain *at least* all the referenced objects.

As mentioned in Section 4.2.2.1, the XML structure is specified in a concrete manner, fixed to this example. Whilst ordinarily the shape of XML data structures will be known in advance, such limitation does negate the benefit of XML being a generic structure. It would be for future work to explore the possibility of adding recursive set types to the system by extending the type-inference and SMT-solver encoding, but this is outside the scope of verification for reflection. Nevertheless, the current method of concrete specifications is not a major burden and gives a stronger verification result.

One thing that may be concerning is the number of annotations/lemmas that are needed for the automated verification. Whilst the majority are for unfolding and folding of predicate instances, there are still a large number which deal with asserting pure facts or performing more complex operations on predicates. It should be re-iterated that those lemmas which are solely assisting the SMT-solver and do not use predicate definitions have been omitted for presentation here. It is generally case that the increased number of annotations necessary here compared with earlier examples is due to the complexity and size of the specifications, and accordingly the size of the symbolic state at each stage of symbolic execution. As with the comment about the execution times, some of the lemmas could be eliminated if the efficiency of the SMT-solver on the given entailments was improved. This is clear because several lemmas are used because a timeout is reached when a large entailment problem arises.

From the perspective of the the human verifier, the specification and verification proved to be both challenging and time consuming. This primarily comes down to two issues. Firstly, the size and complexity of specifications that must be devised and written. This includes the number of predicate definitions that must be recalled to ensure their correct usage. One of the large tasks that the specification process entails is asserting that everything is well-defined. In this application such task is more difficult because it must be done for several levels: *(a)* ensuring that all object fields are referencing extant objects, *(b)* ensuring that all objects are well-defined in terms of the metadata, *(c)* ensuring that objects are well-defined in terms of the XML data, and *(d)* ensuring that the XML data is well-defined in terms of objects' fields referencing objects which must also be in the XML. The second main issue effecting the specification is the recursive nature of the program in question. Because some of the procedures sit in mutual recursion, see Figures 4.18 and 4.28, the specifications almost need to represent the greatest-common-denominator and include sufficient formulae to support all the other procedures. This means that a change to one procedure's specification will often necessitate a similar change to at least one other procedure, which itself may "break" the verification in another way and require another alteration of the specifications. The complex specifications are therefore not user-friendly for the average programmer to produce, and involves a significant time investment. It would certainly be worthwhile investigating whether there are ways to make the specifications more manageable without weakening, which can possibly be achieved by extending the assertion language, such as allowing lists rather than just sets.

The first example, of the visitor pattern, is a good case of how the verification can work for generic programs.

Chapter 5

Enhancing reasoning for reflection with the antiframe rule

The antiframe rule [21], first mentioned in Chapter 1 and tool support described in Chapter 2, allows the hiding of "local state". In Chapters 3 and 4 a technique was presented for verifying programs which use reflection. Vital to the approach was the storage of the metadata on the heap such that a reflective library could be implemented. It is the intention that the metadata is only ever "touched" directly by the library itself, and the programs which use reflection should only access the metadata via the library's members. Indeed, it is undesirable for a program to alter the metadata because this could create a situation where the metadata no longer correctly represents the actual program structure.

Due to the fact that the library should be the only entity that directly interacts with the metadata on the heap, it is clear that we have a scenario with a form of local state. It is therefore a useful application for the antiframe rule, such that the rest of the program does not need to be concerned with the metadata. The approach described in this chapter will "move" the library procedures on to the heap such that they may be effected by the antiframe rule when applied to the library/metadata generation process.

While the work in earlier chapters has already provided a usable solution to verifying reflective programs, in this chapter it is shown how an alternative approach using the antiframe rule can be devised.

This chapter first demonstrates how the advanced higher-order store features of the Crowfoot tool can be used to manually create and verify a model of the new approach. Secondly, it is shown how the result can be achieved automatically by building the additional library loading and anti-frame steps in to the tool.

5.1 Introduction

To highlight the problem of metadata corruption on verification, a simple program can be conceived that unfolds the metadata predicates to expose the contents, and then adds a

new method. This new method would need to fulfil the "generic" specification for reflectable methods (see Chapter 3), however it would not be part of the actual declared class. A later part of the program may then look up this "imposter" method with reflection, and then invoke it. The verification of this will succeed as memory safety is still preserved because the new method fits the specifications of the other methods. It may be expected that the method lookup and invocation should fail to verify, because the method was not part of the program structure.

This is the important aspect: the metadata no longer correctly represents the input program. It should be noted that a key idea in the support for verification of reflective programs in this thesis is that Java programs would be appropriately transformed into the Crowfoot language to be verified. As such, because the Java language does not have direct access to metadata itself, the transformed version would not be accessing the metadata anyway. However, for the principles of verification it is a reasonable expectation that the Crowfoot model of reflection is itself correct in behaviour.

There are several ways of preventing metadata corruption. The simplest would be to restrict the unfolding of metadata predicates in annotations for procedures that are not part of the library. This can be done by the parser and would solve any unauthorized tampering, however the metadata is still visible in the heap which might seem redundant to the lay person if it can never be directly used. Another is to add the ability to mark parts of a pre-condition as being immutable, as in [75]. This is good, however the immutability does not restrict the manual inspection of the metadata, which is not possible in Java.

The solution presented here uses the antiframe rule to hide the metadata to programs that use reflection, making it a local resource for the library only. This more closely represents the Java system, where the metadata is not accessible by the program.

The crucial point with the antiframe rule is that the invariant is *local*. In the procedural language here, the metadata is not really local because it is only local to the collective reflective library procedures, but there is no real concept of modules here. Therefore the metadata is not initially local to some definable scope in the logic. For this reason, the *library procedures will be moved on to the heap* so the local scope can be this loading process.

With the central thought in mind that the local invariant to be hidden is the metadata, the first step is to consider in what way the antiframe rule can be applied. First, it is described how the new library-metadata setup can be implemented (and verified) manually using the existing language. Following this, as with the metadata generation, it is shown how this process can be automated with the added benefit of properly representing the structure of the input program.

5.2 Implementation

This section makes use of the predicates describing the metadata, which were defined in Chapter 3. Another crucial aspect is the manner in which predicates can be defined using

$$\mathsf{const}\ f_1;\ \ldots\ \mathsf{const}\ f_n; \quad // \text{ Constant for each library procedure name}$$

$$\mathsf{recdef}\ \$\mathrm{Meta}(;\%M) := \ldots \quad // \text{ Metadata structures as before}$$

$$\mathsf{recdef}\ \$\mathrm{RefLib}(;\%M) := \bigstar_{i=1}^{n} f_i \mapsto \forall \vec{v}_i.\ \{P_i\}\ \_(\vec{t_i})\ \{Q_i\}$$

$$\mathsf{recdef}\ \$\mathrm{RefLibFull}(;\%M) := \$\mathrm{RefLib}(;\%M) \circ \$\mathrm{Meta}(;\%M)$$

$$\mathsf{recdef}\ \$\mathrm{RefInit}() := f_1 \mapsto \_ \star \ldots \star f_n \mapsto \_$$

Figure 5.1: For reflective library procedures $f_1, \ldots, f_n$

the invariant extension operator $\circ$:

$$\mathsf{recdef}\ \$\mathrm{S}(\vec{a}) := \$\mathrm{R}(\vec{b}) \circ \Psi$$

which will $\star$-conjoin the invariant Ψ onto the definition of \$R, and also adding it as an invariant to all nested Hoare-triples.

For the original metadata generation in Chapter 3, the result depended on the structure of the main input program. Here, because the library is to be created on the heap in addition to the metadata, the reflection initialisation step also depends on the structure of the library module. Specifically it depends on the public reflective library procedures, such as those members of Class and Method. The library, to the outside world, will consist of these procedures on the heap *without the metadata*. The result that will be achieved here is that instead of the \$Meta predicate being created in the main program, a new predicate is created, which represents the collection of library procedures. The new predicates are given in Figure 5.1.

First, constants are declared for each of the n library procedures (for instance Field_get, Class_getMethod etc.), using the same name. These will be the addresses of each library procedure on the heap that can be used to invoke them. Next, to describe all the library procedures a recursive predicate \$RefLib is defined which contains these procedures with the "external" metadata-free specifications. Note that these specifications are altered slightly from the original library procedure specifications in terms of the \$Meta instance being replaced by \$RefLib, and the quantified variable for the metadata set ($\%M$). This variable is removed from the sets of universally quantified variables for each loaded version of procedure, so that it is instead bound by the variable in the \$RefLib predicate's arguments. This means that every library procedure is bound to the same metadata. A portion of the concrete definition for \$RefLib would be as follows:

$$
\begin{array}{l}
\mathsf{const}\ \mathit{Class\_getName};\\
\mathsf{recdef}\ \$\mathrm{RefLib}(;\%M) :=\\
\qquad\qquad\vdots\\
\quad \star\ \mathit{Class\_getName} \mapsto\\
\quad \forall\ \mathit{clsPtr}, \mathit{res}, @\mathit{clsName}, \%\mathit{clsMs}, \%\mathit{clsFs}, \mathit{construct}.\\
\quad \{\$\mathrm{RefLib}(;\%M) \star (\mathit{clsPtr}, @\mathit{clsName}, \%\mathit{clsFs}, \%\mathit{clsMs}, \mathit{construct}) \in \%M\\
\qquad\qquad \star\ \mathit{res} \mapsto \_\}\\
\quad \_(\mathit{clsPtr}, \mathit{res})\\
\quad \{\$\mathrm{RefLib}(;\%M) \star \mathit{res} \mapsto @\mathit{clsName}\}\\
\qquad\qquad\vdots
\end{array}
$$

Next, using the $\circ$ operator, a new predicate $RefLibFull is defined from the first, where the metadata is included at the top-level, and deepframed into each specification. The result of framing $Meta onto $RefLib is that every procedure specification in the new predicate definition will then have $RefLibFull in place of $RefLib, which matches the specifications of the procedures as-implemented in the library as will be seen below. The heap data structures for the metadata, which include $Meta, $ClassLseg, $MethodLseg, $FieldLseg, remain unchanged.

The appearance of the $RefLib and $RefLibFull predicates is somewhat unusual, when only the $Meta is actually needed in the library specifications because there is (presently) no recursion in the implementation of the top-level library procedures. The reason that they are needed is that the implementation of Crowfoot currently only handles a special form of user-defined predicate when using the $\circ$ operator. This was to simplify the automation, where a new version of the predicate must be generated without the $\circ$ operator. The restriction requires that the left-hand side of the $\circ$ must appear recursively inside all nested triples of its definition. It is theoretically possible, as shown in [52], for this to be generalised. If this generalised version was implemented in future, the library specifications could be altered to remove the unnecessary library.

Given this new arrangement of predicates for describing the library, and the original fixed procedures, it must be considered how to initially create that library. A predicate is defined to describe the initial state of the heap, $RefInit, where memory must be allocated for each procedure. In a similar way to the appearance of $(\mathit{META} \mapsto 0)$ in the pre-condition of the main procedure acting as the trigger for loading the reflection library (as in Chapter 3), the trigger now additionally includes the occurrence of $RefInit().

Note that in order to fulfil the slightly different arrangement, the original library procedure specifications must also be altered to include this new $RefLibFull predicate, instead of $Meta which it contains. The code bodies will therefore need to be surrounded by an additional unfold and fold annotation in order to expose the $Meta instance again. The

```
proc Class_getName(clsPtr, res)
∀ %M, @clsName, %clsMs, %clsFs, construct.
  pre : $RefLibFull(; %M) ⋆ res ↦ _
        ⋆ (clsPtr, @clsName, %clsFs, %clsMs, construct) ∈ %M;
  post : $RefLibFull(; %M) ⋆ res ↦ @clsName;
{
  ghost 'unfold $RefLibFull(; %M)';
    ...   // Rest of body unchanged
  ghost 'fold $RefLibFull(; %M)';
}
```

Figure 5.2: "Internal" specification for one reflection library procedure

rest of the specifications and hint annotations can remain unchanged. As an example, one procedure is given in Figure 5.2.

In Figure 5.3, it is shown how the reflective library can be set up programmatically, using the existing language features which provide support for the antiframe rule. The procedure *load_reflection* represents the process that is to be automated in the next section, but here shows how the automation can be seen as a macro for the necessary set of operations. The abstract *load_meta* abbreviates the generation of the metadata in the original manner described in Chapter 3.

The body of *load_reflection* first unfolds the predicate containing initialised pointers for each library procedure, and then creates the metadata. Next, the antiframe annotation is used to declare the $Meta predicate to be a local resource that can be hidden outside this procedure. Note that the skolemized variable names (with an integer suffix e.g. *fooPtr0*, *fooCons0*) will not be known until the example is processed by Crowfoot the first time – the correct integers to use are easily identified from the verification output. Each library procedure is then stored in the cell pointed to by a same-named constant, and the collection of loaded procedures and the metadata $Meta can be folded into the single instance of $RefLibFull. The post-condition of the procedure is therefore the version of $RefLibFull without the $Meta invariant.

The *main* procedure is not part of the implicit library "module", but of the utilising program. Here it is seen that the reflection is now invoked through the use of eval commands, rather than call. This trivial change would need to take place to every other use of reflection. Of course, the fixed procedure versions of the library procedures are still available, however the symbolic execution will not be able to proceed with invocations of these because the metadata is required by those specifications, which is no longer present in the generated $RefLib predicate that is now available.

By successfully verifying the *load_reflection* procedure, a degree of soundness is gained about using the antiframe rule with the reflective library and metadata in this way. The

```
proc load_reflection()
   pre : $RefInit() ⋆ META ↦ 0;
   post : ∃%M. $RefLib(;%M);
{
   ghost 'unfold $RefInit()';
   ghost 'antiframe $Meta(... ∪ {(fooPtr0,"foo",%fooFs0,%fooMs0,fooCons0)} ∪ ...)';
   call load_meta();   // generates metadata (lists) on heap
   // Where each f_i is a member of reflective library:
   [f1] := f1(_,...);
    ⋮
   [fn] := fn(_,...);
   ghost 'fold $RefLibFull()';
}
// Models the behaviour of the built-in metadata generation.
proc abstract load_meta()
   pre : meta ↦ _;
   post : ∃fooPtr,%fooFs,%fooMs,fooCons,....
            $Meta(;... ∪ {(fooPtr,"foo",%fooFs,%fooMs,fooCons)} ∪ ...);

proc main()
   pre : $RefInit() ⋆ META ↦ 0;
   post : ∃%M. $RefLib(;%M);
{
   call load_reflection();
   ⋮
   eval [f4](x,y)   // ... do some reflection (previously call f4(x,y) )
}
```

Figure 5.3: Annotated program demonstrating manual usage of the antiframe rule to set up the reflective library

```
proc main(res)
  pre : $RefInit() ⋆ META ↦ 0 ⋆ res ↦ _;
  post : ∃ %M. $RefLib(; %M) ⋆ res ↦ "Foo";
{
   locals obj, cls;
   call Foo_construct(res);
   obj := [res];
   eval [Object_getClass](obj, res);
   cls := [res];
   eval [Class_getName](cls, res)
   call Foo_dispose(obj);
}
```

Figure 5.4: Using the antiframe version of the reflective library, with metadata automatically generated

next section takes the method further by implementing an extension to Crowfoot where the metadata is automatically generated and the library automatically loaded on to the heap.

5.3 Integrating with metadata generation

The goal of automating the approach above will result in the ability to make use of reflection as in Figure 5.4, where some class Foo is present.

Let *RefProcs*(Γ) be a function that filters (by string pattern matching) a set of fixed procedures in Γ identified as being interfaces to the reflective library by the prefix of their name:

$$RefProcs(\Gamma) = \{(\mathsf{proc}\ \mathcal{F}(\vec{z})\ldots) \in \Gamma \mid \text{matches}(\mathcal{F}, MetaClasses)\}$$

where

$$MetaClasses = \{\text{`Object\_*'}, \text{`Class\_*'}, \text{`Method\_*'}, \text{`Field\_*'}, \text{`Constructor\_*'}\}$$

This set contains all of the library procedures that are to be stored on the heap.

There are now four elements that must be generated for the automated setting up of the reflection library:

1. The metadata argument of $Meta.
2. The code for loading each procedure in *RefProcs*(Γ) onto the heap (in a similar way to *load_reflection*).
3. The definition of $RefInit.
4. The definition of $RefLib.

The metadata generation algorithm *generateMeta* can proceed as described in Chapter 3, Section 3.5, to produce an appropriate set representation of the metadata.

$$
\begin{array}{rcl}
\mathit{createLib}(\Gamma) & = & \mathsf{ghost}\ \text{`}\mathsf{unfold}\ \$\mathrm{RefInit}(;?)\text{'};\\
& & \mathit{createLoadStmts}(\Gamma);\\
& & \mathsf{ghost}\ \text{`}\mathsf{fold}\ \$\mathrm{RefLibFull}(;?)\text{'}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{createLoadStmts}\big(()\big) & = & \mathsf{skip}\\
\mathit{createLoadStmts}\big((\mathsf{proc}\ \mathcal{F}(\vec{x})\ldots),\Gamma\big) & = & [\mathcal{F}] := \mathcal{F}(\vec{\_});\ \mathit{createLoadStmts}(\Gamma)\\
& & \text{where } |\vec{x}| = |\vec{\_}|
\end{array}
$$

$$
\begin{array}{l}
\mathsf{proc}\ \mathcal{F}_{load\_refl}()\\
\quad \mathsf{pre} : \$\mathrm{RefInit}() \star \mathit{META} \mapsto 0;\\
\quad \mathsf{post} : \$\mathrm{RefLibFull}(\ \alpha\ );\\
\{\ C\ \}\\
\text{where } C = \mathit{createLib}(\mathit{RefProcs}(\Gamma))\\
\text{and } \alpha = \mathit{generateMeta}(\Pi,\Gamma)
\end{array}
$$

Figure 5.5: Function generating store-code statements for each reflective library procedure, and the generated procedure that loads the library

For generating the code that will load the metadata, a function *createLib* is defined in Figure 5.5, which takes a sequence of procedures and generates a sequence of code statements. The resulting code is slightly different to the code of *load_reflection* given in Figure 5.3 in two respects. Firstly, there is no call to *load_meta*(), because the metadata is automatically added to the symbolic state. Secondly, the antiframe is not applied at this stage. Instead, the antiframe rule will be applied later to the procedure itself before being invoked at the start of the main body. By using this function, a procedure declaration $\mathcal{F}_{load\_refl}$ can be generated which will perform the loading of the reflective library.

The definition of the $RefInit predicate, which is a list of initialised pointers, can be trivially created based on the library procedure names. The definition of the $RefLib predicate, which needs to use specifications in Γ as well as the names, can be created along similar lines however it is complicated by the fact that the specifications needed in the predicate definition are not identical to the declared fixed procedures. The specifications in the predicate must a) all use the same metadata set argument, bound by the argument of $RefLib, and b) should remove the metadata invariant from the specifications. However the changes can by made trivially through syntactic matching. The fact that the syntactic changes are correct will be proved automatically because the generated code that loads the library is verified like any other procedure. This will ensure that the specifications in $RefLib properly represent a deepframe-subtracted version of the full "internal" specifications. Definitions for both these predicates are given in Figure 5.6.

These two predicates are added to the predicate context Π, and the generated loading

$$\mathsf{recdef}\ \$\mathrm{RefInit}() := \overset{n}{\underset{i=1}{\bigstar}} f_i \mapsto \_; \quad \text{for } (\mathsf{proc}\ f_1 \ldots), \ \ldots, \ (\mathsf{proc}\ f_n \ldots) \in \mathit{RefProcs}(\Gamma)$$

$$\mathsf{recdef}\ \$\mathrm{RefLib}(;\%M) :=$$
$$\overset{n}{\underset{i=1}{\bigstar}} f_i \mapsto \forall \vec{v}. \begin{array}{c} \{\ (\exists \vec{w}_1.\$\mathrm{RefLib}(\%M) \star \Phi_1) \vee \ldots \vee (\exists \vec{w}_n.\$\mathrm{RefLib}(\%M) \star \Phi_n)\ \} \\ f_i(\vec{p}_i) \\ \{\ (\exists \vec{w'}_1.\$\mathrm{RefLib}(\%M) \star \Phi'_1) \vee \ldots \vee (\exists \vec{w'}_n.\$\mathrm{RefLib}(\%M) \star \Phi'_n)\ \} \end{array};$$

for each $f_i \in \mathit{RefProcs}(\Gamma)$ where:

$$\forall \%M, \vec{v}. \begin{array}{c} \{\ (\exists \vec{w}_1.\$\mathrm{RefLibFull}(\%M) \star \Phi_1) \vee \ldots \vee (\exists \vec{w}_n.\$\mathrm{RefLibFull}(\%M) \star \Phi_n)\ \} \\ f_i(\vec{p}_i) \\ \{\ (\exists \vec{w'}_1.\$\mathrm{RefLibFull}(\%M) \star \Phi'_1) \vee \ldots \vee (\exists \vec{w'}_n.\$\mathrm{RefLibFull}(\%M) \star \Phi'_n)\ \} \end{array}$$

Figure 5.6: Generated definitions for predicates in the reflective library

procedure $\mathcal{F}_{load\_refl}$ is added to the procedure context Γ so that it will be verified. A new rule is now defined for handling the main procedure where the reflective library is set up, which is used instead of the original in Chapter 3. This rule, in Figure 5.7, essentially replaces the initial $RefInit() and $META \mapsto 0$ in the pre-condition with the loaded version of the library, after the metadata has been hidden. The main procedure is then verified with this new pre-condition.

There now needs to be a special handling of the generated $\mathcal{F}_{load\_refl}()$ procedure, because the metadata gets automatically generated and inserted onto the heap before this procedure executes. In Chapter 3, the metadata generation was instead taking place at the beginning of $\mathcal{F}_{main}()$. The rule for this, LoadRefLibWithMeta, is essentially an instance of MainWithMeta where the procedure name is changed.

With these new rules and the generated predicate and procedure declarations, a program that uses reflection can be verified. The program's main procedure will be given the reflective library in its pre-condition, contained in the predicate $RefLib, and may proceed as in the earlier Figure 5.4.

5.4 Soundness

To show the soundness of the rule (MainReflectionAntiframe), the operational semantics must be extended in a similar manner to the metadata generation step in Chapter 3. The new cases are in Figure 5.8. Recall that h^{γ}_{meta} is the heap representation of the metadata.

The first case is the original semantics for call from [52], with the addition of the condition that if the metadata pointer is in the heap, then it should have been initialised

$$\textsc{MainReflectionAntiframe}\quad \frac{\begin{array}{c}\big(\$\text{RefInit}() \Leftrightarrow f_1 \mapsto \_ \star \ldots \star f_n \mapsto \_\big) \in \Pi \\ \big(\$\text{RefLibFull}(\beta) \Leftrightarrow (\$\text{RefLib}(\beta) \circ \$\text{Meta}(\beta))\big) \in \Pi \\ \{\$\text{RefInit}() \star \mathit{META} \mapsto 0\}\, \mathcal{F}_{\mathit{load\_refl}}()\, \{\$\text{RefLibFull}(\alpha)\} \in \Gamma \\ \Pi; \Gamma \Vdash \{\Psi \star \$\text{RefLib}(\alpha)\}\, \mathcal{F}_{\mathit{main}}(\mathit{params}(\mathcal{F}_{\mathit{main}}))\, \{Q\}\end{array}}{\Pi; \Gamma \Vdash \{\Psi \star \$\text{RefInit}() \star \mathit{META} \mapsto 0\}\, \mathcal{F}_{\mathit{main}}(\mathit{params}(\mathcal{F}_{\mathit{main}}))\, \{Q\}}$$

$$\alpha = \mathit{generateMeta}(\Pi, \Gamma)$$

$$\textsc{LoadRefLibWithMeta}\quad \frac{\Pi; \Gamma \Vdash \{\$\text{RefInit}() \star \$\text{Meta}(\alpha)\}\, \mathcal{F}_{\mathit{load\_refl}}()\, \{\$\text{RefLibFull}(\alpha)\}}{\Pi; \Gamma \Vdash \{\$\text{RefInit}() \star \mathit{META} \mapsto 0\}\, \mathcal{F}_{\mathit{load\_refl}}()\, \{\$\text{RefLibFull}(\alpha)\}}$$

$$\alpha = \mathit{generateMeta}(\Pi, \Gamma)$$

Figure 5.7: New rule for verifying the main procedure with pre-condition extended by metadata

$$\begin{array}{rcl}
(\mathsf{call}\ \mathcal{F}(e_1, \ldots, e_n),\ s \cdot \eta,\ h) & \leadsto^{\gamma} & (C; \mathsf{return},\ s \cdot \eta \cdot \eta[x_1 \mapsto [\![e_1]\!]_\eta, \ldots, x_n \mapsto [\![e_n]\!]_\eta, \\
& & \qquad y_1 \mapsto 0, \ldots, y_m \mapsto 0],\ h) \\
& & \text{if } \mathcal{F} \in \mathsf{dom}(\gamma) \text{ and} \\
& & \quad \gamma(\mathcal{F}) = \mathsf{proc}\,(x_1, \ldots x_n)\{\mathsf{locals}\, y_1, \ldots, y_m; C\} \\
& & \text{and } (\mathcal{F} \notin \{\mathcal{F}_{\mathit{load_refl}}, \mathcal{F}_{\mathit{main}}\} \\
& & \qquad \text{or } (\mathit{META} \in \mathsf{dom}(h) \text{ and } h(\mathit{META}) \neq 0)) \\
(\mathsf{call}\ \mathcal{F}_{\mathit{main}}(\vec{e}), s \cdot \eta, h) & \leadsto & (\mathsf{call}\ \mathcal{F}_{\mathit{load_refl}}(); \mathsf{call}\ \mathcal{F}_{\mathit{main}}(\vec{e}), s \cdot \eta, h) \\
& & \text{if } h = h' \cdot [\mathit{META} \mapsto 0] \\
(\mathsf{call}\ \mathcal{F}_{\mathit{load_refl}}(), s \cdot \eta, h) & \leadsto & (\mathsf{call}\ \mathcal{F}_{\mathit{load_refl}}(), s \cdot \eta, h' \cdot h_{\mathit{meta}}\gamma) \\
& & \text{if } h = h' \cdot [\mathit{META} \mapsto 0]
\end{array}$$

Figure 5.8: Extensions to the operational semantics

and not point to 0. Otherwise the second case applies where the metadata has not been initialised. This triggers the (generated) library loading procedure to be invoked before executing main. In contrast to the special handling of main before, where the metadata was added to the heap, the metadata is now added to the heap before the call to the library loading procedure *refl_lib*. This is fulfilled by the third case.

The soundness of the two new rules now follows.

Theorem 15 (Soundness of LoadRefLibWithMeta)**.** *Proof.* Using the modified operational semantics above, the soundness is as with the soundness of the original rule (MainWithMeta) in Theorem 14, where the procedure $\mathcal{F}_{\mathit{main}}$ is substituted by $\mathcal{F}_{\mathit{load\_refl}}$, and Ψ by \$RefInit(), and Q by \$RefLibFull(α). □

Theorem 16 (Soundness of MainReflectionAntiframe)**.** *The soundness involves first applying the antiframe rule to the procedure in the context* Γ*, and then using that*

with the new operational semantics and sequential composition to show the conclusion.

Proof. First, the antiframe rule is applied to the $\mathcal{F}_{load\_refl}$ in the third premise as follows.

Assume $\pi \models \Pi$. Then we have that $\$\text{RefInit}() \Leftrightarrow f_1 \mapsto \_ \star \ldots \star f_n \mapsto \_$. With the fact that $a \mapsto \_ \otimes I \Leftrightarrow a \mapsto \_$ and distribution of $\otimes$ through $\star$:

$$\$\text{RefInit}() \star \mathit{META} \mapsto 0 \quad \Leftrightarrow \quad (\$\text{RefInit}() \star \mathit{META} \mapsto 0) \otimes \$\text{Meta}(\alpha) \tag{5.1}$$

By the third premise, and the fact that

$\big(\$\text{RefLibFull}(\beta) \Leftrightarrow (\$\text{RefLib}(\beta) \circ \$\text{Meta}(\beta))\big)$ we have:

$$\{\$\text{RefInit}() \star \mathit{META} \mapsto 0\}\, \mathcal{F}_{load\_refl}()\, \{\$\text{RefLib}(\alpha) \circ \$\text{Meta}(\alpha)\}$$

which, by (5.1) is equivalent to:

$$\{(\$\text{RefInit}() \star \mathit{META} \mapsto 0) \otimes \$\text{Meta}(\alpha)\}\, \mathcal{F}_{load\_refl}()\, \{\$\text{RefLib}(\alpha) \circ \$\text{Meta}(\alpha)\} \tag{5.2}$$

Assume an n and $\Gamma \models_\pi \gamma$ such that

$$\gamma \models^n_\pi \{(\$\text{RefInit}() \star \mathit{META} \mapsto 0) \otimes \$\text{Meta}(\alpha)\}\, \mathcal{F}_{load\_refl}()\, \{\$\text{RefLib}(\alpha) \circ \$\text{Meta}(\alpha)\}$$

Then one has, by definition:

$$n \models^\gamma_\pi ((\$\text{RefInit}() \star \mathit{META} \mapsto 0) \otimes \$\text{Meta}(\alpha),\ \mathsf{call}\ \mathcal{F}_{load\_refl}(),\ \$\text{RefLib}(\alpha) \circ \$\text{Meta}(\alpha)) \tag{5.3}$$

Take the following instance of the antiframe rule in Theorem 9:

if $\models^\gamma_\pi$
$((\$\text{RefInit}() \star \mathit{META} \mapsto 0) \otimes \$\text{Meta}(\alpha), \mathsf{call}\ \mathcal{F}_{load\_refl}(), \$\text{RefLib}(\alpha) \circ \$\text{Meta}(\alpha))$
then $\models^\gamma_\pi (\$\text{RefInit}() \star \mathit{META} \mapsto 0,\ \mathsf{call}\ \mathcal{F}_{load\_refl}(),\ \$\text{RefLib}(\alpha))$

By applying this to (5.3), we get the antiframe's conclusion. Using the frame rule (ShallowFrame) with Ψ as the invariant (because there are no global variables or mutable parameters, it can be assumed that the side condition holds $\mathit{mod}(\mathsf{call}\ \mathcal{F}_{load\_refl}()) \cap \mathit{fv}(\Psi) = \emptyset$), we get:

$$\models^\gamma_\pi (\$\text{RefInit}() \star \mathit{META} \mapsto 0 \star \Psi,\ \mathsf{call}\ \mathcal{F}_{load\_refl}(),\ \$\text{RefLib}(\alpha) \star \Psi) \tag{5.4}$$

By the fourth premise, we have:

$$\models^\gamma_\pi (\Psi \star \$\text{RefLib}(\alpha),\ \mathsf{call}\ \mathcal{F}_{main}(\mathit{params}(\mathcal{F}_{main})),\ Q) \tag{5.5}$$

Using (5.4) and (5.5) with the rule of sequential composition (SCOMP), we get:

$$\models^{\gamma}_{\pi} (\$\text{RefInit}() \star \mathit{META} \mapsto 0 \star \Psi,\ \mathsf{call}\ \mathcal{F}_{\mathit{load\_refl}}(); \mathsf{call}\ \mathcal{F}_{\mathit{main}}(\mathit{params}(\mathcal{F}_{\mathit{main}})),\ Q)$$

or equivalently, for all k, η, σ, w, k:

$$w, \eta, \sigma \models^{\gamma}_{k} (\llbracket \$\text{RefInit}() \star \mathit{META} \mapsto 0 \star \Psi \rrbracket,\ \mathsf{call}\ \mathcal{F}_{\mathit{load\_refl}}(); \mathsf{call}\ \mathcal{F}_{\mathit{main}}(\mathit{params}(\mathcal{F}_{\mathit{main}})),\ \llbracket Q \rrbracket)$$

By Definition 6, the above is equivalent to: for all $r \in \mathit{UPred}(H), m < n, h, s$, if $(m, \eta, \sigma, h) \in \llbracket \$\text{RefInit}() \star \mathit{META} \mapsto 0 \star \Psi \rrbracket_{\pi}(w) \star i^{-1}(w)(\mathsf{emp}) \star r$ then

$$(\mathsf{call}\ \mathcal{F}_{\mathit{load\_refl}}(); \mathsf{call}\ \mathcal{F}_{\mathit{main}}(\mathit{params}(\mathcal{F}_{\mathit{main}})),\ s \cdot \eta, h) \in \text{Safe}^{\gamma}_{m} \tag{5.6}$$

$$\begin{array}{l} \text{For all } k \leq m, h', \eta', \\ \text{if } (\mathsf{call}\ \mathcal{F}_{\mathit{load\_refl}}(); \mathsf{call}\ \mathcal{F}_{\mathit{main}}(\mathit{params}(\mathcal{F}_{\mathit{main}})),\ s \cdot \eta, h) \rightsquigarrow^{\gamma}_{k} (\mathsf{skip}, s \cdot \eta', h') \\ \text{then } (m - k, \eta', \sigma, h') \in \bigcup_{w'} \llbracket Q \rrbracket_{\pi}(w \circ w') \star i^{-1}(w \circ w')(\mathsf{emp}) \star r \end{array} \tag{5.7}$$

To show the conclusion, it is required to prove:

$$\models^{\gamma}_{\pi} (\Psi \star \$\text{RefInit}() \star \mathit{META} \mapsto 0,\ \mathsf{call}\ \mathcal{F}_{\mathit{main}}(\mathit{params}(\mathcal{F}_{\mathit{main}})),\ Q)$$

or equivalently, for all k

$$k \models^{\gamma}_{\pi} (\Psi \star \$\text{RefInit}() \star \mathit{META} \mapsto 0,\ \mathsf{call}\ \mathcal{F}_{\mathit{main}}(\mathit{params}(\mathcal{F}_{\mathit{main}})),\ Q)$$

or equivalently, for all η, σ, w, k

$$w, \eta, \sigma \models^{\gamma}_{k} (\llbracket \Psi \star \$\text{RefInit}() \star \mathit{META} \mapsto 0 \rrbracket_{\pi},\ \mathsf{call}\ \mathcal{F}_{\mathit{main}}(\mathit{params}(\mathcal{F}_{\mathit{main}})),\ \llbracket Q \rrbracket_{\pi})$$

By Definition 6, it therefore must be shown that for all $r \in \mathit{UPred}(H), m < n, h, s$, if $(m, \eta, \sigma, h) \in \llbracket \Psi \star \$\text{RefInit}() \star \mathit{META} \mapsto 0 \rrbracket_{\pi}(w) \star i^{-1}(w)(\mathsf{emp}) \star r$ then

(a) $(\mathsf{call}\ \mathcal{F}_{\mathit{main}}(\mathit{params}(\mathcal{F}_{\mathit{main}})),\ s \cdot \eta, h) \in \text{Safe}^{\gamma}_{m}$

(b) For all $k \leq m, h', \eta'$, if $(\mathsf{call}\ \mathcal{F}_{\mathit{main}}(\mathit{params}(\mathcal{F}_{\mathit{main}})),\ s \cdot \eta, h) \rightsquigarrow^{\gamma}_{k} (\mathsf{skip}, s \cdot \eta', h')$ then $(m - k, \eta', \sigma, h') \in \bigcup_{w'} \llbracket Q \rrbracket_{\pi}(w \circ w') \star i^{-1}(w \circ w')(\mathsf{emp}) \star r$

By the cases for call in the new operational semantics in Figure 5.8, the first case does not apply to the above because we have $\mathcal{F}_{\mathit{main}}$ and $h \in \llbracket \mathit{META} \mapsto 0 \rrbracket$ (by the interpretation of $\star$), which by the interpretation of assertions means $h(\mathit{META}) = 0$. The second case does apply and therefore to show (a) suffices to show:

$$(\mathsf{call}\ \mathcal{F}_{\mathit{load\_refl}}(); \mathsf{call}\ \mathcal{F}_{\mathit{main}}(\mathit{params}(\mathcal{F}_{\mathit{main}})),\ s \cdot \eta, h) \in \text{Safe}^{\gamma}_{m-1} \tag{5.8}$$

and for (b):

$$\begin{array}{l}\text{For all } k \leq m-1, h', \eta', \\ \text{if } (\mathsf{call}\ \mathcal{F}_{load\_refl}(); \mathsf{call}\ \mathcal{F}_{main}(params(\mathcal{F}_{main})),\ s \cdot \eta, h) \rightsquigarrow_k^{\gamma} (\mathsf{skip}, s \cdot \eta', h') \\ \text{then } (m-1-k, \eta', \sigma, h') \in [\![Q]\!]_{\pi}(w) \star i^{-1}(w)(\mathsf{emp}) \star r\end{array} \tag{5.9}$$

By the result of the sequential composition above, which gave (5.6) and (5.7), instantiating m with $m-1$ gives the required (5.8) and (5.9)

□

5.5 Discussion

This chapter has shown how Pottier's antiframe rule [21] can be useful in hiding the local state belonging to the reflective library, which is the metadata. The result is that programs which use reflection cannot directly access the metadata and do not need to carry it in the specifications. This has the benefit of more closely representing the Java model where the metadata is not accessible, and also ensuring that programs do not corrupt the metadata.

Whilst the metadata has been hidden from the heap, the programs now have to carry the library procedures in their specifications. One may think therefore that nothing has been saved in the size of specifications as one heap predicate has just been replaced by another, which is true. However the new scenario is more intuitive: when the metadata was present it was never used by a program outside the library. Here, with the antiframe setup, the *library* is in the heap instead, which *is* directly used by programs.

To use the reflection library now, unfortunately there is the unusual occurrence of eval commands rather than calls as would normally be used in the translation from the Java-like setting. This is not a major issue because any systematic translation can easily identify the reflective library procedures and use the eval instead. Additionally the idea of loading the reflection library, as with `import java.lang.reflect.*` in Java, it should not be unnatural to expect the loaded library to have been loaded on the heap.

There is a disadvantage with the approach here, in terms of the specifications of the library procedures. They now must include the predicate $RefLibFull, which consists of the metadata *as well as all of the reflective library procedures on the heap*, regardless of whether they will be used or not. This is done because the set representation of the metadata is still important and is contained in the predicate's argument.

This "antiframe" approach can be considered an optional alternative to the "original" in Chapter 3, as suited to an individual's tastes. The original approach has the advantage in that the library procedures are ordinary fixed procedures, and are not on the heap. The specifications of the library procedures are also simpler in the original implementation because they did not additionally include the rest of the reflective library and did not need an additional unfolding of the $RefLibFull predicate in order to expose the metadata. However these changes are trivial to comprehend, and the antiframe approach is more natural and better representing the Java model. There is no additional benefit in expressivity by

using the new antiframe approach, because the same metadata structure is used in each case. Common in both cases is that the predicates that are carried around by programs that use reflection (either $Meta or $RefLib) do not need to be manually unfolded during verification as Crowfoot automatically unfolds predicate instance by one level if the triple sought for an eval is not explicitly contained in the symbolic heap.

A possible extension to consider is to use the generalized antiframe rule [59], which allows the invariant to "evolve". This could be useful if the reflection library was extended to allow dynamic loading of new classes, where the metadata may grow. This is not currently supported because the applications of reflection that have been focussed on have been for generic programs where the full code is known in advance. However dynamic loading can be verified statically to some extent if it is known what is being loaded in terms of behaviour [76], proving that the new code works with an old program. The new invariant for the generalized antiframe could use the $\subseteq$ relation, and while most library procedures will maintain the same metadata, if there was a procedure that loads a new class and forces the metadata to be updated appropriately, then the new metadata will be a superset of the old.

This extension is dependent on how useful it would be to have the ability to perform class loading in a static verification context, which is questionable. There are two cases of loading to consider. First, if loading new *unknown* classes could take place then the specifications would not be able to say anything useful about the loaded code, and similarly the additional metadata would be unknown. This could, however, be useful for a system with support for "proof-carrying code" [77], where unknown code is only loaded that meets a given specification declared by the original program. In such circumstances, the behaviour of the loaded code would be known. The second case would be for loading new classes that *are known* at verification time. Here the loading can be given a proper specification (and be verified), and the additional content of the metadata can be described. However this type of program could also be statically verified by assuming all the loadable parts have been loaded at the start.

Related work that has been considered is the idea of "immutability", where separation logic formulae can be annotated as immutable [75]. For instance, the metadata predicate could be annotated with $@I$, the flag for signifying an immutable formula, $\$\mathrm{Meta}(;\%M)@I$. This prevents the possibility of programs corrupting the metadata, and could additionally be used in the specifications of the library for those procedures that do not modify the metadata. In fact, because there is no class loading or unloading none of the library procedures need to modify the metadata. Presently the library methods may alter the metadata so long as the content is maintained, so the order of the lists can be altered. Therefore the immutable annotation would make the specifications more precise. However, this technique does not tackle the motivation for the antiframe rule, where the metadata does not need to be accessed by non-library procedures, not even in a read-only fashion. It could still be useful to add the immutability constraint to the library procedures which would work in combination with the antiframe rule.

As a side note, Pottier mentions later how the antiframe rule is "paranoid" and its use restricted when it comes to the using libraries in the type-capability system [78]. This was due to the presence of $P \otimes I$ in the pre-condition. Consider the case if P contains some library procedures that perform common linked list operations, and say I is a linked list that should be hidden. Then the library code will end up having a duplicate capability I as a result of applying the $\otimes$ operator. This paranoia does not occur here because the library procedures are not included in the pre-condition, rather they are created inside the scope of the antiframe.

Chapter 6

Conclusions & Future work

This thesis has presented a method for verification of programs that use Java-like reflection. The approach makes use of a logic with nested Hoare triples which is important for ensuring that the behaviour of reflectively invoked code is consistent with the accompanying metadata describing the program. This is the first such solution supporting static verification of reflective programs where the behaviour about reflective operations is reasoned in conjunction with accompanying metadata. This guarantees that the result of calling some reflective operation is based on the actual metadata description of the program.

In order for a program to be reflective, there must be a representation of itself that is accessible. This representation is the metadata and it is typically stored on the heap. The separation logic foundation was therefore important in providing reasoning about the metadata structures on the heap, as well as being important for describing object instances. The core structures for the metadata were standard linked lists. These have the benefit of being widely understood and therefore it should be straight-forward for the work to be extended or adapted, without needing specific knowledge about a new logic for reflection.

Similarly, the reflective library has been implemented using standard heap manipulation commands that perform traditional linked-list traversal operations. This means that the reflective operations could be verified using existing proof rules. The alternative approach would be to add the reflective operations to the programming language, and devise new proof rules to support them. In this situation the metadata might be contained in a new context that the proof rules depend on.

The extension of separation logic for higher order store with nested triples proved an elegant solution to allowing strong behavioural specifications to be written that describe code that is being invoked reflectively. This was because the nested triples are able to use variables from the outside assertion. When placed in the metadata, a nested triple may inherit variables from the metadata structure in which it is present. There are two operations that perform reflective invocation after inspecting the structure of a class: 1) creating a new instance of an object, and 2) invoking a method. The behaviour of the reflective constructor ensures that the fields of the resulting object are consistent with the

fields described in the metadata for the given class. The behaviour of method invocations is dependent on the types of its arguments, which is also described by the metadata. The correct link between the method's actual argument types and the metadata allows verification to ensure that a Java-style argument type exception is avoided.

In order to support reflection, several enhancements were made to the verification system and tool that was developed for higher-order store reasoning with nested triples. These extensions have applications outside of reasoning for reflection and have contributed to making the tool more powerful allowing the expression of stronger specifications.

One extension implemented in the tool was the for supporting the antiframe rule [21]. This is the first tool to support this kind of antiframing where local state may be hidden. A later version of the reflective library explored using the antiframe rule to hide the metadata from client programs. The antiframe rule proved beneficial in this respect and this alternative system for verifying reflection better represented the Java model where the metadata is not accessible to the program.

This thesis included two case studies that were inspired by real programs. They served to facilitate a thorough trial of the proposed model for a reflective library. Undertaking the verification of these examples highlighted the importance of maintaining a state of "well-definedness" such that all objects are instances of classes that are in the metadata. The specifications of these programs were large and complicated, especially in the case of the serialization example. The specifications were therefore hard to understand, although they demonstrated that the assertion language, which includes fairly simple formula, is sufficiently expressive and flexible to be able to describe complex functional properties. The main issue with the serialization example was that in addition to objects being well-defined with respect to the metadata, the XML structure also needed to be well-defined with both itself and the metadata.

The specifications of the library itself are not overly-complicated however, which is good for the person writing specifications for a reflective program because it is clear what properties must be fulfilled. Additionally, writing specifications requires no knowledge of the underlying data structures in which the metadata is stored.

The verification time for the case studies by the Crowfoot tool is a concern. The reflective library alone can take nearly a minute to verify. The examples where large symbolic heaps build up take far longer due to the complex entailments that arise. The bottleneck however seems to be the SMT solver. The symbolic execution steps performed by Crowfoot do not take long, even when handling large states. When using Crowfoot with the Yices solver [56] the verification time is much quicker. However, as was discussed in Chapter 2 the Yices verifier is not able to prove the same number of entailments as Z3 [55].

6.1 Comparisons to related work

There has been little work in supporting verification for the type of reflection described in this thesis. Whilst there are several systems aimed at supporting verification of Java programs, they mostly ignore the reflection API. The system that comes closest is the Java Modeling Language (JML).

JML [53] is a specification language for Java programs. Automated verification of JML specifications can be undertaken by tools such as ESC/Java2 [79] or KRAKATOA [80]. The former tool is not sound or complete, a sacrifice that was made in favour of usability considerations. Many of the key Java API classes have been specified such that library classes may be used in programs that are verified. However, some of these specifications are weak, including those for the reflection classes.

Specifications in JML include a pre- and post-condition and more advanced assertions such as class invariants and exception handling. Additionally, specifications may use special JML classes, which include descriptions of common structures such as sets. Methods may be annotated as "pure", to signify that they have no side-effects. The specifications of the classes pertinent to reflection are very often simply annotated as pure, or as having a non-null return value. JML does not have access to a form of metadata, which partly explains the weakness of specifications.

The only method with an equivalent strength specification to that here is for Object.getClass(). This uses an auxiliary variable *_getClass* which is assigned to the result of the special type-of operator, which returns the dynamic class of the given object. Additionally the specification states that the return value is not null. This specification is essentially the same as that in this thesis.

```
//@ public model non_null Class _getClass;
//@ public represents _getClass <- \typeof(this);

/*@ public normal_behavior
  @   ensures \result == _getClass;
  @   ensures_redundantly \result != null;
  @*/
public /*@ pure @*/ final /*@non_null*/ Class getClass();
```

The other notable specification is that of Method.invoke, however the specification simply asserts that if a primitive is returned, then it is wrapped up into the appropriate wrapper class. The specification makes no assertions about the behaviour of the method being reflectively invoked, which can be done using the techniques in this thesis with definitions of the $Pre and $Post predicates.

The weakness of the JML specifications is to some extent likely due to lack of effort, as reflection is not one of the most commonly used features in Java and priority would be best given to common API features such as lists and arrays. Whilst there is no representation

of metadata, however, it would not be possible to write specifications as strong as those in this thesis. Overall the specifications of the reflective library in Chapter 3 are much more precise and meaningful than the current published JML specifications.

One of the limitations of this work is the very simple encoding of an object-oriented system. Reasoning about object-oriented programming has been explored elsewhere with success and was not the focus of this work. The jStar [35] tool, which uses abstract predicate families to reason about inheritance with method overriding and specialization, also uses separation logic and directly handles the Java language. These abstract predicates are already closely related to the mechanism used to specify and verify the method invoke procedure.

Another related tool is VeriFast [42] whose approach to supporting object-oriented principles with "instance predicates" is very similar to abstract predicate families. This tool also uses the Java language, in addition to C, and provides support for function pointers. The approach here could be implemented in VeriFast with the metadata represented in the same heap structures.

VeriFast also inspired the extension to Crowfoot for lemma procedures. The same approach was used here where the bodies may only include side-effect free statements, and they are verified in the same way as procedures. However, their lemmas additionally check for termination which ensures that reasoning over inductive structures is sound. In the Crowfoot system, the human verifier must be careful that inductive proofs are executed in the right way, which will involve consulting the proof graph outputs. VeriFast's proof of termination identifies patterns of induction, and limit to only direct recursion. This is a limitation that is not present when the lemma proofs are "free".

The reflection supported in this work has not included dynamic loading of new code. Static reasoning is difficult for these programs due to the possibly unknown nature of new code. Whilst it would be possible to statically verify the loading process for a particular known instance, see [81] and [76], there is other work in allowing a program to inspect the specifications of loaded code. This is known as Proof-Carrying Code (PCC) [77]. A bytecode version of JML was proposed in [82] that would support PCC, however this does not seem to have been realised.

6.2 Future work

The reflective library does not represent the complete set of reflective operations offered in the Java API. The addition of further operations, however, is limited by the simple model of an object-oriented language, which lacks handling of attributes such as access modifiers or inheritance. Future work will involve seeing what further Java-like reflective features can be supported without large changes to support stronger object-oriented reasoning. It is then hoped to create a specified library by a similar approach in either VeriFast or jStar where the object-oriented paradigm is better supported.

In work on static verification of dynamic updates, a technique for merging an old program with an update is given as a transformation [81]. This transformation is proved sound by a bisimulation argument that shows that the semantics of the merged program are equivalent to the original program with the update applied. It would be interesting to explore whether a similar argument can be made that formalises the translation of a reflective Java program into the Crowfoot language where the entire program is needed statically.

One of the extensions to the Crowfoot system was adding support for reasoning with the antiframe rule [21]. The implementation is limited to the standard antiframe rule, and not the fully generalized version proposed later [59]. Future work could include implementing the generalized version. This would entail modifying the $\otimes$ distribution and invariant subtraction rules to handle the ternary tensor, and make use of the $\leq$ and $\subseteq$ relations that are already in Crowfoot's language. Invariants can already use existential quantifiers, and so the implementation should not be a large body of work.

It would be a worthwhile task exploring ways to improve the efficiency of the verification by tackling the SMT solver bottleneck. The encoding of Crowfoot entailments is not particularly intelligent, and it is likely that there are more efficient encodings for certain patterns of formulae. For instance, if there is a set expression such as $(a, b, \%C) \in \%S$ and the properties of the nested set $\%C$ are not described in the rest of the assertion, then the variable $\%C$, which is encoded as a function, could be substituted for an integer variable. This would mean that the type of $\%S$ is not in the sets-of-sets class as far as the SMT solver is aware. With more knowledge of the implementation of the heuristics inside the solvers, it may be possible to have Crowfoot first normalize assertions in some way that makes them easier for the SMT solver to process.

Bibliography

[1] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, 1969. 2

[2] C. A. R. Hoare, "Procedures and parameters: An axiomatic approach," in *Symposium on Semantics of Algorithmic Languages* (E. Engeler, ed.), vol. 188 of *Lecture Notes in Mathematics*, pp. 102–116, Springer Berlin / Heidelberg, 1971. 2

[3] E. D. Berger and B. G. Zorn, "Diehard: Probabilistic memory safety for unsafe languages," in *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, (New York, NY, USA), pp. 158–168, ACM, 2006. 3

[4] P. W. O'Hearn, J. C. Reynolds, and H. Yang, "Local reasoning about programs that alter data structures," in *Proceedings of the 15th International Workshop on Computer Science Logic*, CSL '01, (London, UK, UK), pp. 1–19, Springer-Verlag, 2001. 3

[5] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *LICS*, pp. 55–74, 2002. 3

[6] J. C. Reynolds, *The Craft of Programming*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1981. 4

[7] P. W. OHearn, "Resources, concurrency, and local reasoning," *Theor. Comput. Sci.*, vol. 375, pp. 271–307, Apr. 2007. 4, 8

[8] P. B. Hansen, "Structured multiprogramming," *Commun. ACM*, vol. 15, pp. 574–578, July 1972. 4

[9] M. Parkinson and G. Bierman, "Separation logic and abstraction," in *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, (New York, NY, USA), pp. 247–258, ACM, 2005. 5, 10, 73

[10] B. Reus and T. Streicher, "About Hoare logics for higher-order store," in *ICALP* (L. Caires, G. F. Italiano, L. Monteiro, C. Palamidessi, and M. Yung, eds.), vol. 3580 of *Lecture Notes in Computer Science*, pp. 1337–1348, Springer, 2005. 5

[11] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux device drivers (third edition)*. O'Reilly Media, 2005. 5

[12] M. Hicks, J. T. Moore, and S. Nettles, "Dynamic software updating," in *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pp. 13–23, ACM, June 2001. 5

[13] P. J. Landin, "The mechanical evaluation of expressions," *Computer Journal*, vol. 6, pp. 308–320, January 1964. 5

[14] B. Reus and J. Schwinghammer, "Separation logic for higher-order store," in *CSL*, pp. 575–590, 2006. 5

[15] K. Honda, N. Yoshida, and M. Berger, "An observationally complete program logic for imperative higher-order functions," in *LICS*, pp. 270–279, 2005. 6

[16] N. Yoshida, K. Honda, and M. Berger, "Logical reasoning for higher-order functions with local state," *Logical Methods in Computer Science*, vol. 4, no. 4, 2008. 6

[17] J. Schwinghammer, L. Birkedal, B. Reus, and H. Yang, "Nested Hoare triples and frame rules for higher-order store," in *CSL*, pp. 440–454, 2009. 6, 14

[18] J. Schwinghammer, L. Birkedal, B. Reus, and H. Yang, "Nested Hoare triples and frame rule for higher-order store," *Logical Methods in Computer Science*, vol. 7, September 2011. 6, 17

[19] N. Charlton and B. Reus, "A deeper understanding of the deep frame axiom." Extended abstract, presented at LOLA (Syntax and Semantics of Low Level Languages), 2010. 6

[20] N. Charlton and B. Reus, "Specification patterns and proofs for recursion through the store," in *FCT*, pp. 310–321, 2011. 7

[21] F. Pottier, "Hiding local state in direct style: a higher-order anti-frame rule," in *LICS*, (Pittsburgh, Pennsylvania), pp. 331–340, June 2008. 7, 23, 35, 36, 37, 63, 146, 158, 162, 165

[22] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang, "Compositional shape analysis by means of bi-abduction," in *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, (New York, NY, USA), pp. 289–300, ACM, 2009. 7, 9

[23] A. Charguéraud and F. Pottier, "Functional translation of a calculus of capabilities," in *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, (New York, NY, USA), pp. 213–224, ACM, 2008. 7

[24] J. Schwinghammer, L. Birkedal, and K. Støvring, "A step-indexed kripke model of hidden state via recursive properties on recursively defined metric spaces," in *Proceedings of the 14th International Conference on Foundations of Software Science and Computational Structures: Part of the Joint European Conferences on Theory and Practice of Software*, FOSSACS'11/ETAPS'11, (Berlin, Heidelberg), pp. 305–319, Springer-Verlag, 2011. 7, 42, 63

[25] L. Birkedal, B. Reus, J. Schwinghammer, K. Støvring, J. Thamsborg, and H. Yang, "Step-indexed Kripke models over recursive worlds," in *POPL'11*, pp. 119–132, IEEE, Jan 2011. 7, 20

[26] J. Schwinghammer, H. Yang, L. Birkedal, F. Pottier, and B. Reus, "A semantic foundation for hidden state," in *FOSSACS*, pp. 2–17, 2010. 7

[27] J. Berdine, C. Calcagno, and P. W. O'Hearn, "A decidable fragment of separation logic," in *Proceedings of the 24th International Conference on Foundations of Software Technology and Theoretical Computer Science*, FSTTCS'04, (Berlin, Heidelberg), pp. 97–109, Springer-Verlag, 2004. 8

[28] J. Berdine, C. Calcagno, and P. W. O'Hearn, "Symbolic execution with separation logic," in *APLAS*, pp. 52–68, 2005. 8, 14

[29] J. Berdine, C. Calcagno, and P. W. O'Hearn, "Smallfoot: Modular automatic assertion checking with separation logic," in *FMCO*, pp. 115–137, 2005. 8

[30] M. Sagiv, T. Reps, and R. Wilhelm, "Parametric shape analysis via 3-valued logic," *ACM Trans. Program. Lang. Syst.*, vol. 24, pp. 217–298, May 2002. 8, 9

[31] A. Møller and M. I. Schwartzbach, "The pointer assertion logic engine," in *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, (New York, NY, USA), pp. 221–231, ACM, 2001. 8

[32] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O'Hearn, "Scalable shape analysis for systems code," in *Proceedings of the 20th International Conference on Computer Aided Verification*, CAV '08, (Berlin, Heidelberg), pp. 385–398, Springer-Verlag, 2008. 9

[33] D. Distefano, P. W. O'Hearn, and H. Yang, "A local shape analysis based on separation logic," in *TACAS*, pp. 287–302, 2006. 9

[34] D. Distefano, "Attacking large industrial code with bi-abductive inference," in *Proceedings of the 14th International Workshop on Formal Methods for Industrial Critical Systems*, FMICS '09, (Berlin, Heidelberg), pp. 1–8, Springer-Verlag, 2009. 10, 22

[35] D. Distefano and M. J. Parkinson, "jStar: towards practical verification for Java," in *OOPSLA*, pp. 213–226, 2008. 10, 164

[36] M. J. Parkinson, *Local reasoning for Java*. PhD thesis, Computer Laboratory, Nov. 2005. 10

[37] M. J. Parkinson and G. M. Bierman, "Separation logic, abstraction and inheritance," in *POPL*, pp. 75–86, 2008. 10, 11, 63

[38] M. Botinčan, D. Distefano, M. Dodds, R. Grigore, and M. J. Parkinson, "corestar: The core of jstar," in *First International Workshop on Intermediate Verification Languages*, BOOGIE, pp. 65–77, 2011. 10

[39] B. Jacobs and F. Piessens, "The verifast program verifier," Tech. Rep. CW-520, Departement Computerwetenschappen, Katholieke Universiteit Leuven, 2008. 10, 27

[40] B. Jacobs, J. Smans, and F. Piessens, "A quick tour of the VeriFast program verifier," in *APLAS*, pp. 304–311, 2010. 10, 53

[41] B. Jacobs, J. Smans, and F. Piessens, "Verification of unloadable modules," in *FM*, pp. 402–416, 2011. 11

[42] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens, "VeriFast: A powerful, sound, predictable, fast verifier for C and Java," in *NASA Formal Methods*, pp. 41–55, 2011. 11, 62, 63, 164

[43] M. Botincan, M. Parkinson, and W. Schulte, "Separation logic verification of c programs with an smt solver," *Electr. Notes Theor. Comput. Sci.*, vol. 254, pp. 5–23, 2009. 11

[44] B. C. Smith, "Reflection and semantics in lisp," in *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '84, (New York, NY, USA), pp. 23–35, ACM, 1984. 11

[45] I. R. Forman and N. Forman, *Java Reflection in Action (In Action series)*. Greenwich, CT, USA: Manning Publications Co., 2004. 12, 112, 114, 116

[46] ANSI X3J13, *Programming Language Common Lisp (formerly ANSI X3.226-1994)*. InterNational Committee for Information Technology Standards, 1994. 12

[47] Oracle, "Java platform SE 6," 2011. `http://docs.oracle.com/javase/6/docs/api/`. 12

[48] ISO/IEC 23270:2006, *Information Technology – Programming Languages – C#*. ISO, 2006. 12

[49] C. Herzeel, P. Costanza, and T. D'Hondt, "Reflection for the masses," in *Self-Sustaining Systems* (R. Hirschfeld and K. Rose, eds.), pp. 87–122, Berlin, Heidelberg: Springer-Verlag, 2008. 13

[50] J. Palsberg and C. B. Jay, "The essence of the visitor pattern," in *Proceedings of the 22nd International Computer Software and Applications Conference*, COMPSAC '98, (Washington, DC, USA), pp. 9–15, IEEE Computer Society, 1998. 14, 100, 133

[51] "The Crowfoot website," 2011. www.sussex.ac.uk/informatics/crowfoot. 14

[52] B. Reus, N. Charlton, and B. Horsfall, "Symbolic execution proofs for higher order store programs," *Journal of Automated Reasoning*, Submitted, 2013. http://sussex.ac.uk/Users/bgh21/crowfoot_jar.pdf. 14, 20, 31, 61, 62, 95, 149, 154

[53] G. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, P. Chalin, D. Zimmerman, and W. Dietl, "JML reference manual (draft v1.235)." Department of Computer Science, Iowa State University. Available from http://www.jmlspecs.org, July 2008. 22, 163

[54] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended static checking for Java," in *PLDI*, pp. 234–245, 2002. 22

[55] L. De Moura and N. 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, (Berlin, Heidelberg), pp. 337–340, Springer-Verlag, 2008. 26, 57, 162

[56] B. Dutertre and L. D. Moura, "The Yices SMT solver," tech. rep., SRI International, 2006. 26, 57, 162

[57] N. Charlton, B. Horsfall, and B. Reus, "Crowfoot: A verifier for higher-order store programs," in *VMCAI* (V. Kuncak and A. Rybalchenko, eds.), vol. 7148 of *Lecture Notes in Computer Science*, pp. 136–151, Springer, 2012. 32

[58] B. Horsfall, N. Charlton, and B. Reus, "Verifying the reflective visitor pattern," in *FtFJP*, pp. 27–34, 2012. 32, 68, 90, 97

[59] F. Pottier, "Generalizing the higher-order frame and anti-frame rules," *Unpublished, July*, 2009. 36, 38, 39, 159, 165

[60] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A modular reusable verifier for object-oriented programs," in *Proceedings of the 4th International Conference on Formal Methods for Components and Objects*, FMCO'05, (Berlin, Heidelberg), pp. 364–387, Springer-Verlag, 2006. 53

[61] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, (Berlin, Heidelberg), pp. 348–370, Springer-Verlag, 2010. 53, 62

[62] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies, "Vcc: A practical system for verifying concurrent c," in *Proceedings of the 22Nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, (Berlin, Heidelberg), pp. 23–42, Springer-Verlag, 2009. 53

[63] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst, "Hampi: a solver for string constraints," in *Proceedings of the eighteenth international symposium on Software testing and analysis*, ISSTA '09, (New York, NY, USA), pp. 105–116, ACM, 2009. 53

[64] Y. Zheng, X. Zhang, and V. Ganesh, "Z3-str: a z3-based string solver for web application analysis," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, (New York, NY, USA), pp. 114–124, ACM, 2013. 53

[65] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A symbolic execution framework for javascript," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, (Washington, DC, USA), pp. 513–528, IEEE Computer Society, 2010. 53

[66] D. Déharbe, "Integration of smt-solvers in b and event-b development environments," *Sci. Comput. Program.*, vol. 78, pp. 310–326, Mar. 2013. 54

[67] L. de Moura and N. Bjorner, "Generalized, efficient array decision procedures," in *Formal Methods in Computer-Aided Design, 2009. FMCAD 2009*, pp. 45–52, 2009. 57

[68] K. R. M. Leino, "Automating induction with an smt solver," in *Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'12, (Berlin, Heidelberg), pp. 315–331, Springer-Verlag, 2012. 62

[69] A. Charguéraud, "Program verification through characteristic formulae," in *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, (New York, NY, USA), pp. 321–332, ACM, 2010. 62

[70] F. Bobot, J.-C. Filliâtre, C. Marché, A. Paskevich, *et al.*, "Why3: Shepherd your herd of provers," in *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pp. 53–64, 2011. 62

[71] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995. 99

[72] R. C. Martin, "The open-closed principle," *More C++ gems*, pp. 97–112, 1996. 100

[73] J. Blosser, "Java tip 98: Reflect on the visitor design pattern," *JavaWorld*, 2000. http://www.javaworld.com/javaworld/javatips/jw-javatip98.html. 100

[74] J. Hunter and B. McLaughlin, "The JDOM project," *Available in http://www.jdom.org*, 2000. 114

[75] C. David and W.-N. Chin, "Immutable specifications for more concise and precise verification," in *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, (New York, NY, USA), pp. 359–374, ACM, 2011. 147, 159

[76] N. Charlton, B. Horsfall, and B. Reus, "Formal reasoning about runtime code update," in *ICDE Workshops* (S. Abiteboul, K. Böhm, C. Koch, and K.-L. Tan, eds.), pp. 134–138, IEEE, 2011. 159, 164

[77] G. C. Necula and P. Lee, "Safe kernel extensions without run-time checking," in *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, OSDI '96, (New York, NY, USA), pp. 229–243, ACM, 1996. 159, 164

[78] F. Pottier, "Three comments on the anti-frame rule." Unpublished, July 2009. 160

[79] D. R. Cok and J. R. Kiniry, "ESC/Java2: Uniting ESC/Java and JML," in *Proceedings of the 2004 International Conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, CASSIS'04, (Berlin, Heidelberg), pp. 108–128, Springer-Verlag, 2005. 163

[80] C. Marché, C. Paulin-Mohring, and X. Urbain, "The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML," *The Journal of Logic and Algebraic Programming*, vol. 58, pp. 89 – 106, 2004. 163

[81] C. M. Hayden, S. Magill, M. Hicks, N. Foster, and J. S. Foster, "Specifying and verifying the correctness of dynamic software updates," in *Proceedings of the International Conference on Verified Software: Theories, Tools, and Experiments (VSTTE)*, January 2012. 164, 165

[82] L. Burdy, M. Huisman, and M. Pavlova, "Preliminary design of bml: A behavioral interface specification language for java bytecode," in *Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering*, FASE'07, (Berlin, Heidelberg), pp. 215–229, Springer-Verlag, 2007. 164