Crowfoot: a verifier for higher order store programs*

Nathaniel Charlton Ben Horsfall Bernhard Reus

School of Informatics, University of Sussex

Abstract. We present Crowfoot, an automatic verification tool for imperative programs that manipulate procedures dynamically at runtime; these programs use a heap that can store not only data but also code (commands or procedures). Such heaps are often called *higher order store*, and allow for instance the creation of recursive procedures on the fly. One can use higher order store to model phenomena such as runtime loading and unloading of code, runtime update of code and runtime code generation. Crowfoot's assertion language, based on separation logic, features *nested Hoare triples* which describe the behaviour of procedures stored on the heap. The tool addresses complex issues like deep frame rules and recursion through the store, and is the first verification tool based on recent developments in the mathematical foundations of Hoare logics with nested triples.

1 Introduction

Dynamic memory that can store not only data but also code is often called *higher order store*. Higher order store thus allows program code to change during execution with the manipulation performed by the program itself. For instance, one may be able to write code onto a mutable heap, invoke it, manipulate it, and then invoke it again later when needed. With higher order store one can model phenomena such as runtime loading and unloading of code — as performed in plugin systems, operating system kernels and dynamic software update systems — and runtime code generation.

Logics with *nested triples* [9, 7], where assertions can contain Hoare triples which describe the behaviour of code stored on the program's heap, have been proposed as a way to reason modularly about higher order store programs. Recent developments [9, 10] have provided solid theoretical foundations for separation logics with nested triples. In this paper we present Crowfoot, the first automatic verification system to apply these developments in practice. Crowfoot has been inspired by previous tools for automated verification using (first order) separation logic, such as Smallfoot [2].

We demonstrate Crowfoot using the program in Fig. 1. The higher order procedure `search` traverses a linked list looking for an element which passes a particular test; `search` takes as an argument a pointer to a stored procedure

* We acknowledge the support of EPSRC grant *"From Reasoning Principles for Function Pointers To Logics for Self-Configuring Programs"* (EP/G003173/1).

```
proc isEql(ptr,val,res,self,i){
 locals tmp;
  tmp := [ptr];
  if tmp = val then {
   if i = 1
   then { [res] := 1 }
   else {
    [self]:= isEql(ptr,_,_,self,i-1);
    [res]:= 0
   }
  } else  { [res] := 0 }
}

proc main(list) {
  locals test, res, value;
  test := new 0; res := new 0;
  value := new 3;
  [test] := isEql(value,_,_,test,2); call search(list, test, res);
  [value]:= 4;
  [test] := isEql(value,_,_,test,1); call search(list, test, res);
  dispose value; dispose test; dispose res
}
```

```
proc search(list,test,res) {
  locals data, next, tmp;
  if list = 0 then {
    [res] := 0
  } else {
    data := [list];
    eval [test](data,res);
    tmp := [res];
    if tmp = 0 then {
      next := [list + 1];
      call search(next,test,res)
    } else {
      [res] := 1
    }}}
```

Fig. 1. Our running example: a higher-order list search program

implementing that test. By using a *self-updating* test procedure, we can use `search` not just to search for elements with particular properties, but also to find out whether a list contains at least a given *number* of such elements.

The first procedure, `isEql(ptr,val,res,self,i)`, essentially compares the content at address `ptr` with the `val` argument. The result is returned in the cell pointed to by `res`. However, there is a twist: whenever `isEql` finds a matching element it will update the content of `self` with a (partially applied) copy of itself, decreasing the counter `i` but returning false. Only when $\texttt{i} = 1$ will it behave like a normal comparison operator. This trick allows one to use the same `search` procedure for different kinds of searches and enables us to easily showcase various Crowfoot features involving stored procedures later. The second procedure, `search(list,test,res)`, performs the list search by recursively traversing the (linked) list starting at address `list`, with the procedure at `test` being applied to each element until 1 (encoding true) is returned. The result is returned in `res`. Finally, `main` shows an example usage of the search procedure, using instances of `isEql` as the test to find 2 and 1 occurrences of values 3 and 4, respectively.

2 Crowfoot's input language

Programming language The programs analysed by Crowfoot are written in an imperative language with recursive procedures, call-by-value parameter passing, and dynamic memory allocation via a mutable heap supporting address arithmetic (to simulate arrays/records) and, crucially, higher order store operations.

integer variables x, set variables α, predicate names P, integer literals n, declared constants c

address expressions	e_A	::=	$x \mid x + n \mid x + c$
value expressions	e_V	::=	$n \mid x \mid e_V + e_V \mid e_V - e_V \mid e_V \times e_V$
element expressions	e_E	::=	$e_V \mid (e_E{}^+)$
set expressions	e_S	::=	$\alpha \mid e_S \cup e_S \mid \{e_E\} \mid \emptyset$
behavioural spec.	B	::=	$\forall [x\|\alpha]^*.\ \{\Phi\} \cdot (x^*) \{\Phi\}$
atomic formula[a]	A	::=	$e_A \mapsto [e_V \mid \_ \mid B]^+$
			$\mid P(e_V{}^*, e_S{}^*) \mid e_V = e_V \mid e_V \neq e_V$
			$\mid e_E \in e_S \mid e_E \notin e_S \mid e_S \subseteq e_S \mid e_S = e_S$
spatial conjunction	C	::=	$\mathsf{emp} \mid A \star C$
assertion	Φ	::=	$\mathit{false} \mid \exists [x\|\alpha]^*.C \ \vee\ \Phi$

[a] We could easily extend this, adding for example intersections or disequalities of sets.

Fig. 2. Abstract syntax for Crowfoot's assertion language

A program is a sequence of declarations, according to the grammar given in Fig. 3. Procedure bodies consist of atomic statements, along with sequential composition, if-then-else conditionals and while loops. These fixed procedures can be loaded onto the heap to obtain *stored procedures*.

Most of the statement forms of the language can be seen in Fig. 1. The `isEql` procedure definition shows the declaration of a local variable (`tmp`), the dereferencing (using `[ ]`) of address `ptr` where the content is stored in `tmp`, and heap assignment operations where 1 or 0 (encoding true or false) is stored in the cell at address `res`. We use the cell at `res` to hold the output because procedures have no return value. Further heap-manipulating statements can be seen in `main` which shows the allocation of three heap cells, with initial content 0, 0 and 3, respectively, and their subsequent disposal at the end of the procedure.

We also have statements for working with higher order store: the `search` procedure uses an `eval` command, which executes the code stored at address `test` with the given arguments. In order for `main` to use the search procedure, the code for the desired test must first be written onto the heap. This is done in lines 5 and 7 of `main`, where we load the implementation of `isEql` into the cell at `test` instantiating some of the arguments at runtime (partial application).

Assertion language Fig. 2 gives Crowfoot's assertion language. We use the logic from [9], restricted to a particular fragment to facilitate automation. The key addition to separation logic is the ability to have *nested triples* within assertions, which allows us to reason about stored procedures. For instance, the assertion $x \mapsto \forall a.\, \{a \mapsto \_\}\ \_(a)\ \{a \mapsto \_\}$ states that address x contains a procedure satisfying Hoare triple $\forall a.\, \{a \mapsto \_\}\ \_(a)\ \{a \mapsto \_\}$; such a procedure runs safely on any argument a which points to an allocated cell ($a \mapsto \_$), and leaves that cell allocated *if* it terminates. We also have set constraints, which do not appear in [9].

Like Smallfoot [2], we do not distinguish between the pure and spatial parts of an assertion, and have only one kind of conjunction operator $\star$; pure formulae

annotated program statements S, (fixed) procedure names $\mathcal{F}$

constant declaration	const c | const $c = n$
predicate declaration	recdef $\mathrm{P}(x^*, \alpha^*) := \Phi$
kringel declaration	kringeldef $\mathrm{P}(x^*, \alpha^*) := \mathrm{P}(x^*, \alpha^*) \circ \exists[x\|\alpha]^*.C$
procedure declaration	proc $\mathcal{F}(x^*)$ $\forall[x\|\alpha]^*$. pre: Φ post: Φ { locals x^*; S }
abstract proc. declaration	proc abstract $\mathcal{F}(x^*)$ $\forall[x\|\alpha]^*$. pre: Φ post: Φ

Fig. 3. Grammar for annotated programs.

such as $x = 0$ are understood to hold only in the empty heap. In input files we write `%a` for set α and `$P` for predicate P; other ASCII notation includes using `|->` for $\mapsto$, $*$ for $\star$, `++` for $\cup$, `<x>` for $\{x\}$, `|` for $\vee$ and `in` for $\in$.

Annotated programs An annotated program is a sequence of declarations, as detailed in Fig. 3. Procedure declarations include behavioural specifications, in the form of pre- and post-conditions. 'Abstract' procedure declarations consist of just the signature and specification; the concrete implementation is omitted. Any `while` statements in procedure bodies are annotated with loop invariants, and `call` statements can have *deep frame* annotations which give invariants to be framed on "deeply". The *deep frame rule* [4, 9], like the regular frame rule of separation logic, adds an invariant I to the pre- and post-condition of a Hoare triple. However, the deep frame rule also adds I as an invariant to all triples nested *inside* the pre- and post-condition (at all levels). User defined predicates are declared via `recdef` and can be inductive (e.g. lists) or recursive. The latter are used for reasoning about recursion through the store [9], e.g. $R(x) := x \mapsto \{P \star R(x)\} \cdot () \{Q \star R(x)\}$. Similarly, declarations with `kringeldef` define predicates with a "deeply" framed on invariant (`@` is ASCII for $\circ$).

3 Specification and verification of the example

We now describe how to use Crowfoot to verify some safety properties of our example program. To describe the linked lists the program uses, we define (using `recdef`) a predicate `$List(x;%a)` denoting a list starting at address `x` and whose node addresses and contents are the pairs in set `%a` (see Fig. 4). As abbreviations we define predicates `$Decider(p)`, stating that `p` points to a two-argument procedure which writes a Boolean to the address in its 2nd argument, and `$FrameDecider(p,x,v)` which is as `$Decider(p)` but with invariant `x|->v` framed on "deeply" (see Fig. 4). Note that `$Decider(p)`, and thus `$FrameDecider(p,x,v)`, are recursive predicates containing "themselves" as invariants, respectively: a decider in `p` requires another (or the same) decider in `p` and ensures that there is another (or the same) decider in `p` after execution.

The pre- and post-conditions we give to the procedures are shown in Fig. 5. For example, `search` requires a list `$List(list; %a)`, a procedure at address `test` as specified by `$Decider`, and an allocated cell at `res`. It ensures that the list has not changed, address `test` still contains a "`$Decider`", and `res` points to a Boolean. As `$Decider(test)` contains a Hoare triple and appears itself inside the pre- and postcondition of `search`, it demonstrates the use of *nested* triples.

```
recdef $List(x;%a) :=  x = 0  *  %a = <>
       |  exists d, n, %b. x |-> d,n * $List(n;%b) *  %a = <(x,d)> ++ %b;

recdef $Decider(p) :=  p |-> forall val,res. {res|->_ * $Decider(p)}
          _(val,res) {exists r. res |-> r * r in <0>++<1> * $Decider(p)};

kringeldef $FrameDecider(p,x,v) :=  $Decider(p) @ x|->v;
```

Fig. 4. Predicate definitions for our example

The `search` procedure requires a test procedure satisfying `$Decider(test)`; such a test procedure uses only one heap cell `res`. However, when `search` is first called in the `main` procedure, `test` points to `isEql(value,_,_,test,2)`; this test procedure, which satisfies `$FrameDecider(test,value,3)`, also uses a heap cell at `value` and thus has a bigger "footprint". To overcome this mismatch we apply the deep frame rule [9] in a statement annotation (explained above):

```
call search(list, test, res) "deepframe value |-> 3";
```

Like Verifast [8], Crowfoot needs annotations at (some) places in the code where it is necessary to fold or unfold user-defined predicates. For instance, in the search procedure, we need to unfold the list predicate at the beginning to be able to see the structure of the list. Then, at the end of the procedure, we need to fold the list back up to meet the postcondition. These steps are achieved with *ghost* statements, where the '`?`' arguments are instantiated by the tool:

`ghost "unfold $List(?;?)";` ... body of procedure... `; ghost "fold $List(?;?)"`

4 The four main parts of the verifier

1. Front end: Crowfoot's front end reads in annotated programs and produces: (1.) an environment of predicate definitions, mapping predicate names to the definitions given to them via the `recdef` or `kringeldef` keywords. (2.) an environment of fixed procedure specifications, mapping the names of fixed procedures to their specifications. (3.) verification conditions (VCs), in the form of one Hoare triple for the body of each fixed procedure, such that if the triples all hold then the program meets its specification. These three outputs are then passed to the symbolic execution engine, which will attempt to prove the VCs.

2. Symbolic execution engine: Crowfoot proves Hoare triples about code using symbolic execution with separation logic, based on ideas put forward in [3] and now well established. The particular challenge in constructing Crowfoot's symbolic execution engine was of course to deal with statements such as `eval` which make use of the higher order nature of the heap. A prover for entailments between assertions is needed at various points during symbolic execution.

3. Entailment provers: Because we work with specifications nested inside assertions, Crowfoot needs *two* (mutually dependent) entailment provers, unlike existing tools: one for proving entailments between assertions, and another for proving entailments between specifications. The SMT solver Yices [6] is used for reasoning about pure formulae (integer and set constraints).

```
proc search(list, test, res)
 forall %a.
   pre: $List(list; %a) * $Decider(test) * res |-> _ ;
  post: exists r.
     $List(list; %a) * $Decider(test) * res |-> r * r in <0>++<1>;  {...}

proc isEql(ptr, val, res, self, i)
 forall v.
   pre: res |-> _ * $FrameDecider(self,ptr,v) ;
  post: i = 1 * val  = v  *  res |-> 1 * $FrameDecider(self,ptr,v)
     |          val != v  *  res |-> 0 * $FrameDecider(self,ptr,v)
     |  i !=1 * val  = v  *  res |-> 0 * $FrameDecider(self,ptr,v); {...}

 proc main(list)
 forall %a.   pre: $List(list; %a);     post:  $List(list; %a);   {...}
```

Fig. 5. Specifications of procedures `search`, `isEql` and `main`

4. Graph output: To aid in the understanding of proofs, and more importantly of failed proof attempts, Crowfoot can produce two kinds of graphical output: trees showing the symbolic execution of procedures, and trees showing proofs (and proof attempts) of entailments.

Crowfoot was written from scratch and consists of ~8.5k lines of OCaml code. Examples we have verified include an idealised updatable web server [5], media player plugin system, and a generic memoiser for recursive functions. Crowfoot can be run on these examples, and the one in this paper, on our website [1].

References

1. The Crowfoot website. `www.informatics.sussex.ac.uk/research/projects/PL4HOStore/crowfoot/`.
2. J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, pages 115–137, 2005.
3. J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. In *APLAS*, pages 52–68, 2005.
4. L. Birkedal, N. Torp-Smith, and H. Yang. Semantics of separation-logic typing and higher-order frame rules for Algol-like languages. *LMCS*, 2(5), 2006.
5. N. Charlton, B. Horsfall, and B. Reus. Formal reasoning about runtime code update. In *HotSWUp (Hot Topics in Software Upgrades), to appear*, 2011.
6. B. Dutertre and L. D. Moura. The Yices SMT solver. Technical report, SRI International, 2006.
7. K. Honda, N. Yoshida, and M. Berger. An observationally complete program logic for imperative higher-order functions. In *LICS*, pages 270–279, 2005.
8. B. Jacobs, J. Smans, and F. Piessens. A quick tour of the VeriFast program verifier. In *APLAS*, pages 304–311, 2010.
9. J. Schwinghammer, L. Birkedal, B. Reus, and H. Yang. Nested Hoare triples and frame rules for higher-order store. In *CSL*, pages 440–454, 2009.
10. J. Schwinghammer, H. Yang, L. Birkedal, F. Pottier, and B. Reus. A semantic foundation for hidden state. In *FOSSACS*, pages 2–17, 2010.