Formal Reasoning about Runtime Code Update

Nathaniel Charlton Ben Horsfall Bernhard Reus

School of Informatics, University of Sussex
Brighton, United Kingdom
{n.a.charlton,b.g.horsfall,bernhard}@sussex.ac.uk

***Abstract*—We show how dynamic software updates can be modelled using a "higher order store" programming language where procedures can be written to the heap. We then show how such updates can be proved correct with a Hoare-calculus that allows for keeping track of behavioural specifications of such stored procedures.**

I. Introduction and motivation

Allowing software to update at run-time is an important feature for critical systems that require constant uptime (see e.g. [1]). But because these are critical systems, it is also very important that updates do not inadvertently crash the system, and that the updated system behaves as expected. There is therefore a need to be able to apply formal methods to the analysis of systems which use dynamic updating.

One mechanism which goes some way towards formally ensuring correctness is type-checking. Bierman *et al.* give a type system for the *update calculus* [2], a functional language featuring modules and an abstract *update* primitive used to model dynamic updates. The type-checking algorithm of their calculus ensures the safety of updates, in other words *well-typed* updates (even those which change the types of functions) do not lead to crashes. Type systems for dynamic update are also discussed in [3] and [4] which address imperative and multi-threaded programs respectively.

Bierman *et al.* also provide a concrete example: a series of runtime updates is applied to an idealised webserver written in the update calculus. Using the type system one can check that the updated system will continue to run safely.

But while type safety is important in ensuring that a program does not crash, we also want to reason about the functional behaviour of a system. This is termed "semantic correctness" in [2]. Such behavioural requirements are typically given in the form of assertions, and pre- and post-conditions for program procedures.

In this paper we present an imperative language with dynamic memory allocation which can also model the webserver example, and show how to prove safety *and* "semantic correctness" properties. The foundation for our language is the ability to update the code stored in memory at runtime. We can then load new code and use code pointers to invoke it. The vital ingredient for reasoning about such languages is the use of *nested Hoare triples* [5], which allow assertions to keep track of behavioural specifications of code stored on the heap; this code changes as programs run due to dynamic updates. Conventional Hoare logics do not provide any mechanism to represent such changes.

We use a semi-automated verification tool, CROWFOOT, to conduct our proofs. The work most closely related to ours is that of [6].

A brief summary of the webserver example is given in Section II. Section III describes how we have modelled the updatable webserver in our language with stored procedures. In Section IV we discuss the specifications of the webserver procedures, including an example for semantic correctness through the addition of an invariant, and their automated proof.

II. Webserver example

In the webserver example of [2], the update mechanism works as follows. Initially, modules are at version 1. When an update takes place, the updated modules are *added* to the current set, and given the next available version number. Before the update is applied, the entire program (with the update) is checked for type-safety. Calls to functions of a module may optionally be qualified with a specific module version; otherwise, the latest version currently loaded will be used. Keeping the old versions and allowing references to specific versions means that the functions may change type during an update.

The single-threaded webserver runs an infinite loop handling event requests from a queue, which, along with standard *get* and *post* requests, includes *update* requests. The initial system consists of two modules:

- Handlers – containing functions to handle the different kinds of event
- Server – containing the loop function which takes an event from the queue and applies the relevant function from the Handlers module, before recursively calling loop again.

Due to space constraints we will only talk about the first dynamic update from [2], although we have also implemented and proved the second [7]. This first update adds logging to the system, so that a record is kept of all the events that have been processed. The changes here involve:

1) the addition of a Log module
2) the handle function to now add each event to the log before calling the appropriate function
3) the loop function to now pass around the log

In order for this to take place, it is necessary for the new version of the Server module to have a transitional loop function, which, when called recursively after the update, will safely start the newly introduced loop′ function with a newly created empty log as the extra log parameter.

In [2] the only correctness property that is checked is type-safety, though that paper then discusses the future possibility of extending the calculus with Hoare-type assertions such that invariants of the loop can express semantic correctness of the program before and after updates.

III. The Webserver implemented with stored procedures

The language we use is an imperative language with recursive procedures, call-by-value parameter passing, and dynamic memory allocation via a mutable heap supporting address arithmetic. The most striking feature is that procedures can be stored on the heap and called at runtime. This is used to load and store modules since our language does not have a built-in update mechanism. The operations for manipulating a heap (where `[e]` is heap lookup at address `e`) are:

```
x    := new 0;     /* Allocate a new heap cell          */
[x] := 5;          /* Content modified to contain 5     */
[x] := foo(_,_);/* We can also store procedures...*/
eval [x](1,2);   /* ... and then invoke them          */
dispose x;         /* Deallocate the cell               */
```

Here we load (or store) a procedure named `foo` into the new cell[1], then call the procedure stored in `x` with parameters before the cell is deallocated. The invocation of stored procedures uses keyword **eval** instead of **call** used for fixed procedures. Should a stored procedure be invoked with the wrong number of arguments, the program will crash. Note that our language is untyped; any type checking will be done as part of the Hoare-logic we describe in the next section.

Having described the features of our programming language, we now present the code of our implementation of the updatable webserver. The shaded elements in Figures 1, 2, 4, and 5 are not part of the code. They can be skipped for now and will be explained in Section IV.

Our language has fixed procedures which can be loaded onto the heap as explained earlier. The loading of the initial modules is done at the start of the `main()` procedure, in Fig. 1. Specifically, `main()` loads the first versions of Server and Handlers and starts the loop with an empty event queue. For the implementation here, we have abstract procedures for manipulating a queue.

The primary difference of our implementation w.r.t. [2] is the use of stored procedures to hold the modules. We have encoded each module using procedures stored in a block of consecutive heap cells, addressed by a constant (declared above `main()`) whose name consists of the module name and version number. For example, version 1 of the Server module is found at address `server1`. Named constant offsets are used to address the particular procedures within each module. A pointer to the latest version of each module is maintained in a cell addressed by a constant without a version number, e.g. `server`.

Next we have the implementations of the Server and Handlers modules in Fig. 2 and 3, respectively. Server's

[1]We can choose to instantiate some of the arguments at load-time (partial application), though this is not necessary for the purposes of the webserver.

```
/* Constants for the addresses of module versions */
/* and offsets of procedures within those modules */
const server1, server2,
  getevent=0, handle=1, loop=2, loopPrime=3;
const handlers1,
  handleGet=0, handlePost=1, handleUpdate=2;
const log1,
  emptylog=0, logEvent=1;
const server, handlers, log; /* 'Latest' pointers */

proc main()
  pre: version ↦ _ * ... ;    post: false;
{
  locals qPtr,q;
  [version] := 1;
  /* Load the first versions of Server & Handlers */
  [server1+getevent] := getevent(_);
  [server1+handle] := handle(_);
  [server1+loop] := loop(_);
  [handlers1+handleGet] := handleGet(_,_);
  [handlers1+handlePost] := handlePost(_,_);
  [handlers1+handleUpdate] := handleUpdate();
  ghost "fold $Code1()";
  /* Set the 'latest' pointers to version 1 */
  [server] := server1;
  [handlers] := handlers1;
  ghost "fold $Code(?)";
  /* Make an empty queue and start the loop */
  qPtr := new 0;
  call mk_empty_queue(qPtr);
  q := [qPtr];
  dispose qPtr;
  eval [server1+loop](q)
}
```

Fig. 1. Initial loading for first version of modules

`handle(q)` and `loop(q)` are implemented as in [2]. In particular, `loop(q)` simply calls the `getevent` and `handle` procedures before recursively calling itself. These procedures are not called directly, however, but via pointers from the heap. This provides "dynamic dispatch"; at runtime new procedures can be added to the heap and the "latest" pointers adjusted accordingly. For the Handlers module, because we are not focussing on how the *get* or *post* requests are handled, we declare the `handleGet(q,req)` and `handlePost(q,req)` procedures as abstract. The `handleUpdate()` simply calls the `update()` procedure, corresponding to the *update* primitive from the update calculus.

The `update()` procedure in Fig. 4 performs the required steps to load the new code into memory, and adjust the "latest" pointers accordingly. In the update considered here, the new Log module (consisting of two procedures) is loaded, along with the new version of Server (see Fig. 5). The Log module consists of abstract procedures for creating an empty log (`mk_empty_log`) and logging an event (`logevent`). The specifications for these and other module procedures can be seen in *$Code2* (Fig. 6). Note that `loop2(_)` is the transitional function, and `loopPrime(_,_)` is the new infinite loop. The type of the new `handle` and `loop` procedures has changed such that they will take a pointer to the log as an additional parameter.

```
/* Constants denoting kinds of events */
const evGet=0, evPost=1, evUpdate=2;

proc abstract getevent(q)

proc handle(q) {
  locals ePtr, event, eventType, req;
  ePtr := new 0;
  call dequeue(q,ePtr);
  event := [ePtr];
  dispose ePtr;
  ghost "unfold $Event(?)";
  eventType := [event];
  req := event+1;
  /* Explicitly call version one of Handlers   */
  /* handleGet, handlePost or handleUpdate     */
  if eventType = evGet then {
    eval [handlers1+handleGet](q,req)
  } else {
    if eventType = evPost then {
      eval [handlers1+handlePost](q,req)
    } else {
      eval [handlers1+handleUpdate]()
    }
  };
  dispose event
}

proc loop(q) {
  locals tmp;
  ghost "unfold $Code(?)";
  /* Call latest version of Server's...   */
  tmp := [server];
  ghost "fold $Code(?)";
  eval [tmp+getevent](q); /* ... getevent */
  ghost "unfold $Code(?)";
  tmp := [server];
  ghost "fold $Code(?)";
  eval [tmp+handle](q);   /* ... handle   */
  ghost "unfold $Code(?)";
  tmp := [server];
  ghost "fold $Code(?)";
  eval [tmp+loop](q)      /* ... loop     */
}
```

Fig. 2. Server module, version 1

```
proc abstract handleGet(q,req)
proc abstract handlePost(q,req)
proc handleUpdate() { call update() }
```

Fig. 3. Handlers module, version 1

IV. Our specification of the Webserver

Our assertion language is an extension of Separation logic [8], a variant of Hoare logic for reasoning about programs that manipulate the heap. Separation logic allows one to specify the behaviour of code in a local way, such that it is only necessary to specify the part of the heap that is manipulated by the code (the footprint), whilst the remaining unspecified portion (the frame) will not change. Its key features are the separating conjunction $\_ * \_$ expressing properties of disjoint parts of the heap, and a predicate $e \mapsto e_1, \ldots, e_N$ stating that e

```
proc update()
  forall ver.
  pre: $Code(ver);
  post: $Code(ver+1) * ver != 2 ∨ $Code(ver) * ver=2; {
  locals v;
  ghost "unfold $Code(?)";

  v := [version];
  if v = 1 then { /* If we didn't update yet */
    [log1+emptyLog] := mk_empty_log(_);
    [log1+logevent] := logevent(_,_);
    [server2+getevent] := getevent(_);
    [server2+handle] := handle2(_,_);
    [server2+loop] := loop2(_);
    [server2+loopPrime] := loopPrime(_,_);
    ghost "fold $Code2()";
    /* Update 'latest' pointers */
    [log] := log1;
    [server] := server2;

    [version] := 2
  } else { /* If we already ran the update */
      skip
  };
  ghost "fold $Code(?)"
}
```

Fig. 4. Procedure modelling the update to add logging

is a pointer to a block of N consecutive heap cells containing values $e_1, \ldots, e_N$ (we abbreviate $\exists n.\, e \mapsto n$ by $e \mapsto \_$).

The crucial extra feature we use is *nested triples* [5]. As explained earlier, it is the use of nested triples that allows us to reason about programs that store procedures (i.e. code) on the heap. In particular, this is how we capture the effects of dynamic updates. For example, the following assertion

$$x \mapsto \forall res.\, \{res \mapsto \_\}\ \_(res)\ \{res \mapsto \_\}$$

states that address x contains a procedure, with one argument res, that satisfies a Hoare triple with the following meaning: if res is allocated when the procedure runs then, if it terminates, the cell is still allocated. As standard with separation logic, triples have a partial correctness, fault-avoiding semantics, so the above triple also states that if res is allocated in the heap the procedure will not crash (even if it does not terminate). The above assertion could be used in pre- or post-conditions of other procedures which leads to *nesting* of triples.

Another addition to our assertion language are user-defined predicates to "wrap-up" assertions. Specifically we use predicates to describe the loaded modules, as shown in Fig. 6. We precede our predicate names with a \$-sign, to distinguish them from ordinary variables in assertions, and define them using the **recdef** keyword. These definitions can be mutually recursive which is important as one needs to be able to specify that a stored procedure invoked via **eval** preserves its own specification and maybe other specifications of procedures it uses. In Fig. 6 we can see that *$Code*(_) is recursively defined via *$Code1* and *$Code2*. We split the specification of the loaded code described by *$Code*(_) into two parts:

recdef *$Code1() :=*

handlers1+handleGet $\mapsto \forall q, req, n.\ \{\$GetRequest(req) * \$Q(q,n)\}\ \_(q, req)\ \{\$Q(q,n)\}$

* *handlers1+handlePost* $\mapsto \forall q, req, n.\ \{\$PostRequest(req) * \$Q(q,n)\}\ \_(q, req)\ \{\$Q(q,n)\}$

* *handlers1+handleUpdate* $\mapsto \forall ver.\ \{\$Code(ver)\}\ \_()\ \{\$Code(ver+1) * ver \neq 2\ \vee\ \$Code(ver) * ver = 2\}$

* *server1+getevent* $\mapsto \forall q, n.\ \{\$Q(q,n)\}\ \_(q)\ \{\$NEmpQ(q,n)\}$

* *server1+handle* $\mapsto \forall q, n.\ \{\$Code(1) * \$NEmpQ(q,n)\}\ \_(q)\ \{\$Code(1) * \$Q(q,n+1)\ \vee\ \$Code(2) * \$Q(q,n+1)\}$

* *server1+loop* $\mapsto \forall q.\ \{\exists n.\$Code(1) * \$Q(q,n)\}\ \_(q)\ \{false\}$;

recdef *$Code2() :=*

log1+emptyLog $\mapsto \forall logPtr.\ \{logPtr \mapsto \_\}\ \_(logPtr)\ \{\exists lg.logPtr \mapsto lg * \$Log(lg,0)\}$

* *log1+logevent* $\mapsto \forall lg,e,n.\ \{\$Log(lg,n) * \$Event(e)\}\ \_(lg,e)\ \{\$Log(lg,n+1) * \$Event(e)\}$

* *server2+getevent* $\mapsto \forall q,n.\ \{\$Q(q,n)\}\ \_(q)\ \{\$NEmpQ(q,n)\}$

* *server2+handle* $\mapsto \forall lg,q,n.\ \{\$Code(2) * \$NEmpQ(q,n) * \$Log(lg,n)\}\ \_(lg,q)\ \{\$Code(2) * \$Q(q,n+1) * \$Log(lg,n+1)\}$

* *server2+loop* $\mapsto \forall q.\ \{\exists n.\$Code(2) * \$Q(q,n)\}\ \_(q)\ \{false\}$

* *server2+loopPrime* $\mapsto \forall lg,q.\ \{\exists n.\$Code(2) * \$Q(q,n) * \$Log(lg,n)\}\ \_(lg,q)\ \{false\}$;

recdef *$Code(v) :=* *(* Encapsulates the entire state of program at version v *)*

*v = 1 * version* $\mapsto$ *1 * $Code1()*

* *handlers* $\mapsto$ *handlers1 * server* $\mapsto$ *server1* *(* Latest versions of Server & Handlers is ver 1 *)*

* *log1* $\mapsto$ *0,0 * log* $\mapsto$ *0 * server2* $\mapsto$ *0,0,0,0* *(* Cells for code that will appear in first update *)*

$\bigvee$

*v = 2 * version* $\mapsto$ *2 * $Code1()*

* *handlers* $\mapsto$ *handlers1* *(* Handlers hasn't changed *)*

* *$Code2()* *(* Code that appeared in the update *)*

* *log* $\mapsto$ *log1 * server* $\mapsto$ *server2* ; *(* Latest Log is ver 1, latest Server is ver 2 *)*

Fig. 6. Predicate definitions used for the webserver specifications, making prominent use of nested triples

- *$Code1* contains the specifications for the initially loaded code, i.e. version 1 of the Server and Handlers modules.
- *$Code2* contains the specifications for procedures that will be added after the update (a new version, 2, of Server, and completely new procedures for version 1 of Log).

Predicate *$Code*($v$) encapsulates the entire state at each stage v of the update process. In this case we only have one update, so we have the initial state (version 1), and the state after the update (version 2). This predicate describes the heap as having the relevant heap cells allocated, and in *$Code1* and *$Code2* we give the behavioural specifications of the loaded code.

Additionally we have (abstract) predicates without definitions (ie. predicate variables), declared in Fig. 7, using the keyword **forall**, to represent the abstract request types and the queue type used. The latter, for instance, is declared abstractly as the predicate *$Q(_,_)*, where the first parameter is the address of the queue, and the second is a counter that increments on each dequeue operation. *$NEmpQ(_,_)* is analogous for non-empty queues. The abstract procedures of our queue datatype are specified as follows: the `dequeue` procedure requires a non-empty queue and ensures a queue, `mk_empty_queue` establishes a queue, and `reset` sets the counter of dequeued elements to 0 and maintains the queue. Finally, the *$Log*(_,_) predicate represents the log structure, with the first parameter being the address of the log and the second parameter being its size.

The program code introduced in Section III has two types of annotation (the shaded elements). The first are the specifications (pre- and post-conditions) for each procedure, and the second are hints to the verifier (***ghost*** statements). The hints instruct the verification tool we use about how and when to fold/unfold predicates. Question marks indicate predicate arguments to be inferred by the verifier. This is necessary for instance at the call site for `handle(q)` in `loop(q)` (Fig. 2). Here we need to "unfold" the *$Code* predicate to be able to find out what is stored in address `[server]`, and we need to "fold" it back up in order that the symbolic state will satisfy the pre-condition when the code at that address is invoked.

Due to space limitations, the specifications of the module procedures have been omitted from the code; they can still be seen in the *$Code1*/*$Code2* predicates. The pre-condition from `main()` in Fig. 1 has been shortened with "...". The missing part specifies that cells have been allocated for every module.

To encode the different kinds of events, we use predicate variables (Fig. 8) combined with a constant integer (Fig. 2) for each kind.

As an example of a behavioural correctness property, we add to our type- and memory safety assertions an invariant that states that the number of entries in the log is the same as the number of events that have been dequeued. This invariant

```
proc abstract mk_empty_log(logPtr)
proc abstract logevent(lg,e)

proc handle2(lg,q) {
  locals ePtr, event, eventType, req, tmp;
  ePtr := new 0;
  call dequeue(q,ePtr);
  event := [ePtr];
  dispose ePtr;

  /* Here is the new behaviour: we log the event. */
  ghost "unfold $Code(?)";
  tmp := [log];
  ghost "fold $Code(?)";
  eval [tmp+logevent](lg,event);
  ghost "unfold $Event(?)";
  eventType := [event];
  req := event+1;
  if eventType = evGet then {
    eval [handlers1+handleGet](q,req)
  } else {
    if eventType = evPost then {
      eval [handlers1+handlePost](q,req)
    } else {
      eval [handlers1+handleUpdate]()
    }
  };
  dispose event
}

proc loop2(q) { /* Transitional loop procedure */
  locals logPtr, newLog;
  logPtr := new 0;
  eval [log1+emptyLog](logPtr);
  newLog := [logPtr];
  dispose logPtr;
  call reset(q); /* Reset queue counter */
  eval [server2+loopPrime](newLog,q)
}

proc loopPrime(lg,q) { /* New loop procedure */
  /* As loop, but evals have extra log parameter */
  ...
}
```

Fig. 5. Code of new Log and Server, added in the update

```
forall $GetRequest(_), $PostRequest(_),
  $Q(_,_), $NEmpQ(_,_),
  $Log(_,_).
```

Fig. 7. Abstract predicates declared without a definition

is supposed to hold every time one re-enters the `loop` body (see occurrence of $\$Q(q,n) * \$Log(lg,n)$ in pre-condition of *loopPrime* in Fig. 6). This n is the reason why we have the second arguments to the *$Log* and *$Q* predicates. Note that all server loops are not supposed to terminate so their postcondition is simply $false$.

In order to keep track of specifications and perform proofs it is extremely helpful to have some software support. We are currently developing a tool called CROWFOOT, which we have used to formalise and prove the specifications as outlined in this paper. CROWFOOT allows one to state programs which use stored procedures and their properties and, once appropriate

```
recdef $Event(e) :=
      e ↦ evGet * $GetRequest(e+1)
    | e ↦ evPost * $PostRequest(e+1)
    | e ↦ evUpdate;
```

Fig. 8. Encoding the event type using predicates

invariants and ghost statements have been sprinkled in, *automatically* verify that all procedures meet their specifications (using ideas extending [9]). More details, including annotated code and outputs for the webserver example, can be found on [7], and publications are in preparation.

V. FUTURE WORK AND CONCLUSION

In this paper we have only discussed the first update of the webserver in [2]. The second, which enriches the event type, has been formalised analogously and proved using CROWFOOT. It can be obtained from [7]. In [2] the loading process is abstractly modelled by a rewrite step, updating the program's runtime configuration that contains all modules loaded over the lifetime of the program; this is modelled in this paper by updating blocks of stored procedures addressed by fixed pointers. A less abstract model of the loading process would use a *linked list* of modules enriched by necessary meta-information that accompanies the modules.[2] Dynamic loading is a form of reflection and it is part of our research programme to develop proof systems for reflective programs including runtime code generation. Finally, we have indicated that our Hoare-logic based approach allows us to prove what [2] calls "semantic correctness". Of course, more complex invariants than the one presented here can be formalised.

ACKNOWLEDGEMENTS

This research has been sponsored by the EPSRC grant "*From Reasoning Principles for Function Pointers To Logics for Self-Configuring Programs*" (EP/G003173/1).

REFERENCES

[1] M. Hicks, "Dynamic software updating," Ph.D. dissertation, Department of Computer and Information Science, University of Pennsylvania, August 2001.

[2] G. Bierman, M. Hicks, P. Sewell, and G. Stoyle, "Formalizing dynamic software updating," in *Proceedings of Workshop on Unexpected Software Evolution (USE '03)*, April 2003.

[3] G. Stoyle, M. W. Hicks, G. M. Bierman, P. Sewell, and I. Neamtiu, "*Mutatis Mutandis*: Safe and predictable dynamic software updating," *ACM Trans. Program. Lang. Syst.*, vol. 29, no. 4, 2007.

[4] A. Anderson and J. Rathke, "Migrating protocols in multi-threaded message-passing systems," in *HotSWUp*, 2009.

[5] J. Schwinghammer, L. Birkedal, B. Reus, and H. Yang, "Nested Hoare triples and frame rules for higher-order store," in *CSL*, 2009, pp. 440–454.

[6] B. Jacobs, J. Smans, and F. Piessens, "Verification of unloadable C modules — soundness proof," Departement Computerwetenschappen, Katholieke Universiteit Leuven, Tech. Rep. CW-570, 2009.

[7] (2010) The Crowfoot website (under development). [Online]. Available: www.informatics.susx.ac.uk/research/projects/PL4HOStore/crowfoot/

[8] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *LICS*, 2002, pp. 55–74.

[9] J. Berdine, C. Calcagno, and P. W. O'Hearn, "Symbolic execution with separation logic," in *APLAS*, 2005, pp. 52–68.

[2]Note that using fixed pointers does not have any impact on the validity of our approach regarding verification which solidly models the one of [2].