Efficient, Formally Verifiable Data Structures using ACL2 Single-Threaded Objects for High-Assurance Systems

David Hardin
Rockwell Collins

Samuel Hardin
Iowa State University

Introduction

- Bounded versions of "Classical" data structures (stacks, queues, deques, etc.) find broad use in high-assurance systems.
- Experience has shown that data structure implementation can result in errors that are often not found until late in the development process.
- Need to prove correctness of data structure implementations at EAL7.
- Challenge to ourselves: Develop an approach for efficient, verifiable data structure implementation, and do a quick (two-week) proof-of-concept.

High-Assurance Development vs. Formal Verification

High Assurance Dev.

- Static programming language subset
- Fixed-size data structures
- No recursion
- Straightforward implementation
- Freedom from exceptions

Formal Verification

- Dynamic programming language subset
- Functional data structures
- Recursion
- Subtle implementation

Verifiable Double-Ended Queues (Deque)

- We wish to implement an efficient formally verifiable, double-ended queue (deque) that conforms to typical high-assurance design rules.
- Functional deque implementations described in the literature (e.g., Okasaki 1995, a portion of which is shown at right in ML) are quite subtle, utilizing a pair of lists, one reversed.
- Lists are of unbounded length, and not appropriate for embedded implementation.
- This style of implementation is not "evaluator-friendly".
- Our solution: Implement the deque using an ACL2 single-threaded object instead.

```
datatype 'a Deque = Deque of {front  : 'a Stream, sizef  : int,
                              rear   : 'a Stream, sizer  : int,
                              pendingf : 'a Stream, pendingr : 'a Stream}

(* INVARIANTS                                                    *)
(*  1. sizef = length front /\ sizer = length rear              *)
(*  2. sizer <= c * sizef + 1 /\ sizef <= c * sizer + 1         *)
(*  3. pendingf <= max (2j+2-k,0) /\                            *)
(*     pendingr <= max (2j+2-k,0)                               *)
(*       where j = min (length front,length rear)               *)
(*             k = max (length front,length rear)               *)
(*                                                              *)
(*  Invariant 3 guarantees that both pending lists are          *)
(*  completed by the time of the next rotation.                 *)

(* In general, c must be greater than or equal to 2, but for *)
(* this implementation we assume c = 2 or 3.  The only place *)
(* this makes a difference is in the function rotate2.       *)
val c = 3

fun tail2 xs = tail (tail xs)

fun take (0,xs) = Stream.empty
  | take (n,xs) = lcons (head xs,fn () => take (n-1,tail xs))

fun drop (0,xs) = xs
  | drop (n,xs) = drop (n-1,tail xs)

fun rotate1 (n,xs,ys) =
    if n >= c then lcons (head xs,
                   fn () => rotate1 (n-c,tail xs,drop (c,ys)))
    else rotate2 (xs,drop (n,ys),Stream.empty)

and rotate2 (xs,ys,rys) =
    (* if c > 3, slightly more complicated code is required here *)
    if Stream.isempty xs then revonto (ys,rys)
    else lcons (head xs,
              fn () =>
                let val (ys,rys) = partialrev (c,ys,rys)
                in rotate2 (tail xs,ys,rys) end)

and revonto (ys,rys) =
    if Stream.isempty ys then rys
    else revonto (tail ys,cons (head ys,rys))

and partialrev (0,ys,rys) = (ys,rys)
  | partialrev (n,ys,rys) = partialrev (n-1,tail ys,cons (head ys,rys)
```

ACL2 Single-Threaded Objects (stobjs)

- ACL2 enforces strict syntactic rules on stobjs to ensure that they are not copied.
- Thus, "old" states of a stobj are guaranteed not to exist.
- This means that ACL2 can provide destructive implementation for stobjs, allowing stobj operations to execute quickly.
- An ACL2 single-threaded object combines a functional semantics about which we can readily reason with a relatively high-speed imperative implementation that more closely follows "normal" design rules for high assurance.

Deque Single-Threaded Object

- Implemented in a way that is typical for a high-assurance developer; many other designs possible.
- Did not declare the deque to be resizable, even though the stobj framework allows this.
- Implemented the deque such that fast block data move instructions could be used when needed.
- To ease reasoning overall, did not use modular arithmetic to maintain the head and tail indexes; rather, used ACL2 to establish that the arithmetic performed on the head and tail indices would never cause an overflow.
- Have implemented other common "classical" data structures in a similar way.

Deque Single-Threaded Object, cont'd.

```
(defstobj dqst
  (arr :type (array t (2048)) :initially (empty))
  (hd  :type (unsigned-byte 11) :initially 0)
  (tl  :type (unsigned-byte 11) :initially 0))
```

- arr size can be increased to million of elements before encountering Lisp limits (using Clozure Common Lisp 1.3).
- empty is a distinguished value that cannot be added to the deque.
- Can vary the initial value of the hd and tl indices depending on the relative amount of adds occurring to the front vs. the back.
- tl index "points" to the last element + 1; hd = tl indicates an empty deque
  - Thus, capacity of the deque is arr size - 1

A Sampling of Deque Operations

- Predicates
  - is-empty, is-full, is-full-front, is-full-back, contains
- Accessors
  - get-first, get-last, size-of
- Mutators
  - add-first, add-last, remove-first, remove-last, clear
  - remove-first-occurrence, remove-last-occurrence

The seq Macro, and an Example of Its Use

```
(defmacro seq (stobj &rest rst)   ;; due to J Moore
  (cond ((endp rst) stobj)
        ((endp (cdr rst)) (car rst))
        (t `(let ((,stobj ,(car rst)))
              (seq ,stobj ,@(cdr rst))))))


(defund remove-last (dqst)
  (declare (xargs :stobjs dqst
                  :guard-hints
                  (("Goal" :in-theory
                           (enable size-of)))))
  (if (= (size-of dqst) 0)
      dqst
      (seq dqst
        (update-tl (- (tl dqst) 1) dqst)
        (update-arri (tl dqst) (empty) dqst))))
```

Deque Theorems

- Proved nearly 60 deque theorems over an approximately two-week period.
- Separated the deque definitions and deque theorems into different files; deque-thms includes the deque-stobj book.
  - Made the hand translation to C, etc., easier by eliminating the "clutter" of defthms not needed to admit the deque operators.
  - Only two lemmas were needed to help admit the deque operators into the logic.

Deque Theorems (cont'd.)

- Proved many correctness results; a number still remain, however.
- Proved some composed results, e.g. that if the deque is not full, then `(equal (get-last (add-last e dqst)) e)`
- Made extensive use of defund, defthmd, and e/d hints
  - In one defthm, needed a stable-under-simplificationp hint to re-enable update-arri.
- Attempted to minimize the number of "hack" lemmas.

A Useful Relation

```
(defun tail-head-relation (dqst)
         (declare (xargs :stobjs dqst))
         (and (natp (hd dqst))
              (natp (tl dqst))
              (<= (hd dqst) (- (maxnode) 1))
              (<= (tl dqst) (- (maxnode) 1))
              (<= (hd dqst) (tl dqst))))
```

We proceed to prove that this relation in fact holds for all deque operations. A sample theorem of this sort:

```
(defthmd add-first--tl-hd--thm
  (implies (tail-head-relation dqst)
           (tail-head-relation (add-first e dqst)))
  :hints (("Goal" :in-theory
           (e/d (add-first size-of is-full
                 is-full-front)
                (update-arri)))))
```

Sample Deque operation correctness theorems

```
(defthm remove-last--sz-correct--thm
  (implies (and (dqstp dqst) (not (= (size-of dqst) 0)))
           (equal (size-of (remove-last dqst)) (- (size-of dqst) 1)))
  :hints (("Goal" :in-theory (enable size-of remove-last))))


(defthmd remove-last--tl-hd--thm
  (implies (tail-head-relation dqst)
           (tail-head-relation (remove-last dqst)))
  :hints (("Goal" :in-theory (e/d (size-of remove-last) (update-
arri)))))


(defthm remove-last--data-correct--thm
  (implies (and (not (is-empty dqst)) (tail-head-relation dqst))
    (equal (update-nth (+ -1 (nth *tl* dqst)) (empty) (nth *arri* dqst))
           (nth *arri* (remove-last dqst))))
  :hints (("Goal" :in-theory (e/d (remove-last size-of is-empty) ()))))
```

Translation to Conventional Programming Languages

- Deque stobj has been hand-translated to C, Java, and SPARK.
- An automated translator could be readily constructed; the biggest challenge would be converting recursion to iteration for languages/compilers that do not support tail recursion elimination.
- Even with a highly automated translation, a "code-to-spec review" would still need to be held with evaluators.
- A code-to-spec review ensures that the specification that is reasoned about (the ACL2 stobj specification) corresponds to the code that is generated.

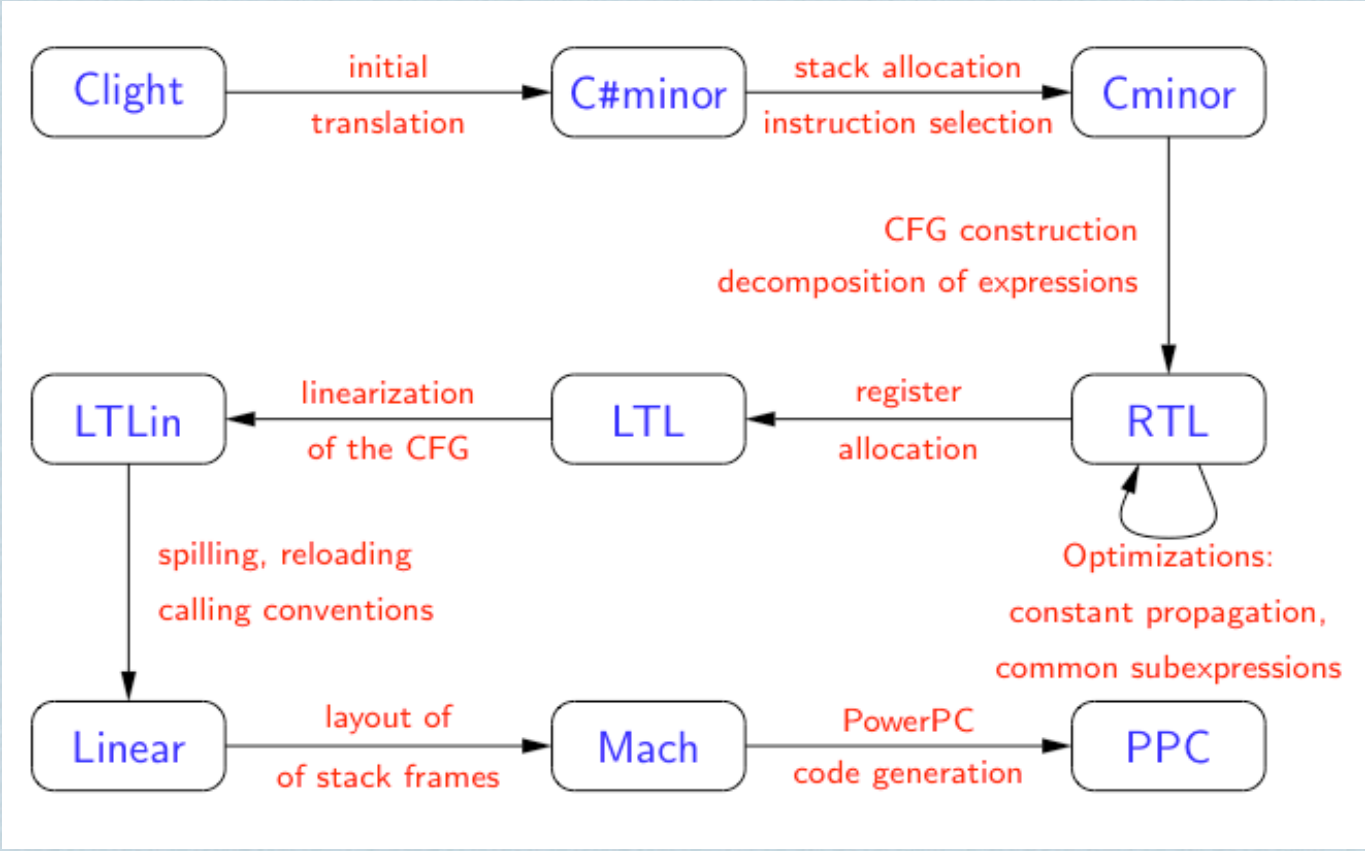
Translated C code sample

```
#define MAXNODE  2048

int *arr[MAXNODE];
int hd = 0;
int tl = 0;

int sizeOf() {
    if (tl > hd) {
        return tl - hd;
    } else {
        return 0;
    }
}

int isEmpty() {
    return hd == tl;
}

int *getFirst() {
    if (isEmpty()) {
        return empty();
    } else {
        return arr[hd];
    }
}
```

A Verified "Stack"

- Once we have the verified data structure translated into a "conventional" source language, it can then be compiled into machine code for execution on an operating system and CPU.
- A verified compiler for a significant subset of C has been developed by Xavier Leroy as part of the CompCert project.
- The CompCert compiler generates code for the PowerPC architecture, which was also the target CPU architecture for the Green Hills INTEGRITY-178B RTOS verification.
- These developments present a new opportunity for a formally verified "stack" (inspired by the "CLInc Stack") from the application layer all the way down to the CPU.
  - Note, however, that the CompCert compiler is verified using the Coq theorem prover. Thus, the verified "stack" described herein is admittedly not as well-integrated as the CLInc Stack.

CompCert C Compiler Passes



CompCert-compiled PowerPC code for getFirst()

```
_getFirst:
    stwu    r1, -64(r1)
    mflr    r2
    stw     r2, 12(r1)
    bl      _isEmpty
    cmpwi   cr0, r3, 0
    bf      2, L102
    addis   r2, 0, ha16(_hd)
    lwz     r3, lo16(_hd)(r2)
    rlwinm  r3, r3, 2, 0xfffffffc
    addis   r2, r3, ha16(_arr)
    lwz     r3, lo16(_arr)(r2)
L103:
    lwz     r2, 12(r1)
    mtlr    r2
    lwz     r1, 0(r1)
    blr
L102:
    bl      _empty
    b       L103
```

Update: CompCert 1.4

- Email exchange with Xavier Leroy in January 2009 confirmed that although the CompCert compiler did identify tail calls, that information was not utilized in code generation.
- On 21 April 2009, after our paper was written, the CompCert project released version 1.4 of the C compiler.
- Notably, the CompCert compiler now includes support for tail recursion elimination.
- Thanks to Xavier Leroy for adding this feature.

Conclusion

- Practical data structures commonly found in high-assurance systems can be specified and verified with a minimum of effort using ACL2 single-threaded objects (stobjs).
- These stobj formulations can be readily translated into high-assurance implementations expressed in conventional programming languages, compiled with a verified compiler, and executed on an EAL6+ operating system platform, thus forming a kind of "verified stack".
- The translation from ACL2 to conventional programming languages has so far been done by hand. Future work will mechanize the translation, and also expand on the types of data structures to be implemented and analyzed.