Using ACL2 in the Design of Efficient, Verifiable Data Structures for High-Assurance Systems

David Hardin and Konrad Slind
Rockwell Collins
Advanced Technology Center

Rockwell Collins

© 2018 Rockwell Collins.
All rights reserved.

Disclaimer

The views expressed are those of the authors and do not reflect the official policy or position of the Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

© 2018 Rockwell Collins. All rights reserved.

Motivation

- Cyber resilience is an increasingly important requirement for Rockwell Collins customers, both government and commercial
- As DARPA HACMS Air Vehicle team lead, we researched methods/tools to create “clean-slate” cyber-resilient systems
- Our air vehicles resisted all attacks by the HACMS red team
- On the new DARPA CASE program, challenges include:
- Making cyber resilience a first-class systems engineering property, on par with the various existing “ilities”
- Applying cyber-resilient engineering methods and tools to systems including significant legacy elements

© 2018 Rockwell Collins. All rights reserved.

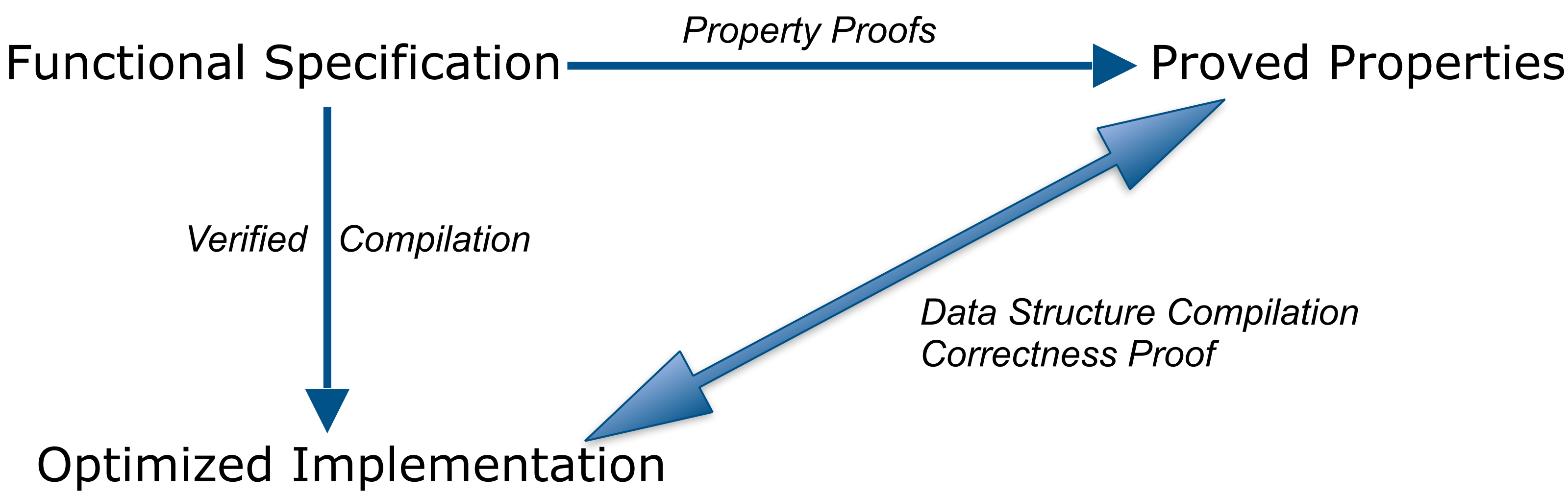
Motivation (cont'd.)

- Additionally, new autonomy functions such as route planning, inference, pattern recognition, etc. present a significant V&V challenge, due to the lack of a human operator, as well as complex new data structures and algorithms
- Proof techniques for these data structures exist, but are oriented to unbounded, functional data types
  - Functional data structure implementations are not often efficient in space or time, so developers generally take a more imperative approach
- We need to find proof techniques that embrace the "natural" functional proof style, yet apply to more efficient data structure implementations
  - Including GPU-based and hardware-based data structures

© 2018 Rockwell Collins. All rights reserved.

Verified Data Structure Compilation and Property Proofs

- Once we develop the Data Structure Compilation Correctness Proof, properties proved of the functional data structure specification will also hold for the optimized implementation



© 2018 Rockwell Collins. All rights reserved.

DASL: A Domain-Aware System Language

- We have developed a "Domain-Aware" System Language, DASL, that embodies our vision:
  - DASL is a system-level language, appropriate for expressing algorithms and data structures that can be compiled to traditional programming languages, GPU languages, as well as Hardware Description Languages (HDLs)
  - HOL4 gives semantics to DASL evaluation, and we use proved source-to-source transformations in HOL4 to compile DASL code
  - DASL is a "mashup" of concepts from Ada, ML, and the C family of languages, and has a similar feel to modern languages such as Swift and Rust

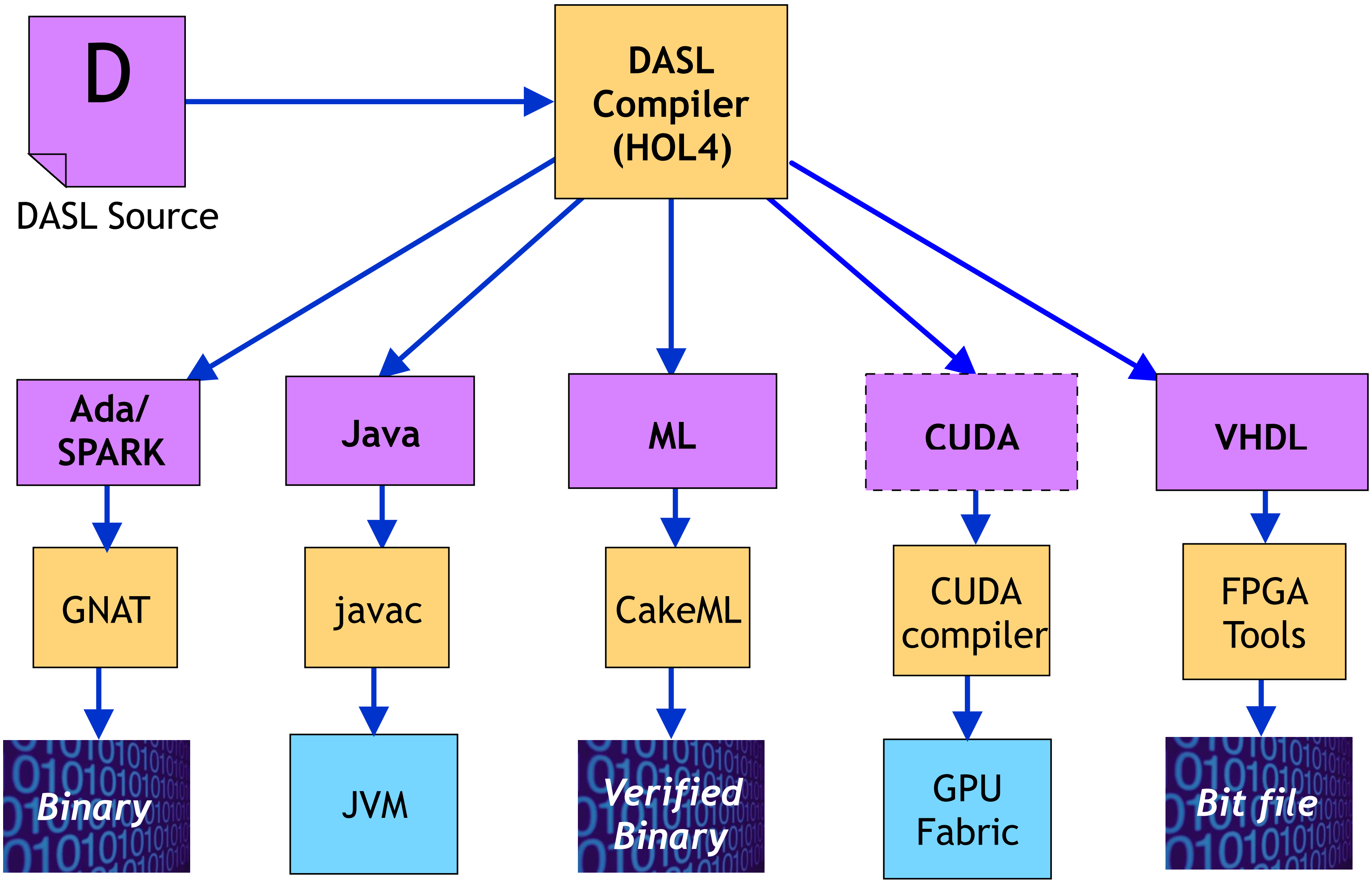
© 2018 Rockwell Collins. All rights reserved.

DASL Data Structure Compilation to Linear Form

- DASL provides ML-like functional data structure specifications
  - Data structure specification includes a maximum size
- DASL compiles Data Structure Specifications into a linearized form requiring no heap allocation or deallocation, in keeping with high-assurance development tenets
  - (e.g. DO-178C Level A)
- The DASL toolchain produces proofs that data structure operations on the compiled form are equivalent to the same operations on the high-level functional form
  - Proves that in-place updates are equivalent to functional (copying) updates, given that no "old" copies of the data structure are allowed
- User-defined properties are introduced using `spec` statements

© 2018 Rockwell Collins. All rights reserved.

DASL Code Generation Options



© 2018 Rockwell Collins. All rights reserved.

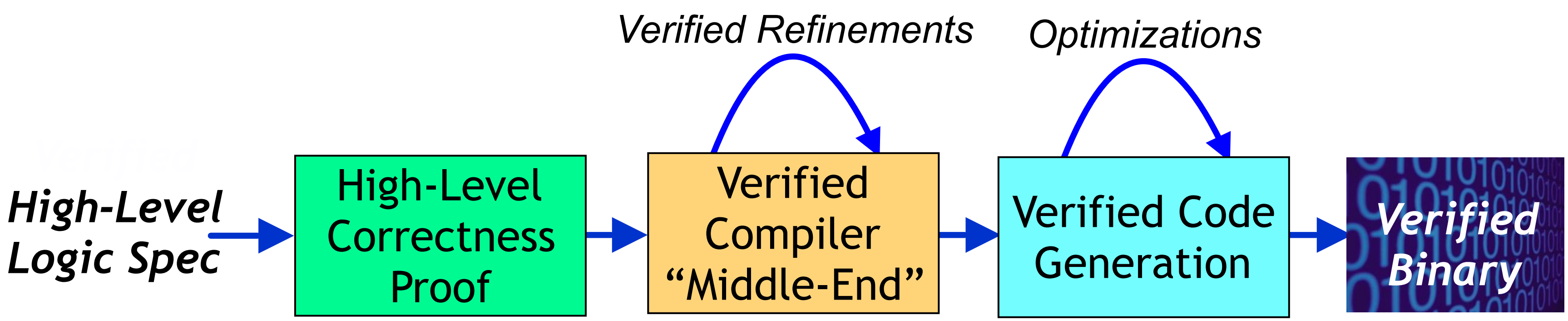
Related Work: Theorem Provers and Verified/ Verifying Compilers

- A number of verified/verifying compilers have emerged recently
  - e.g., CompCert C compiler, CakeML
- Concurrently, theorem provers have increasingly supported the translation of logic functions to code, e.g.:
  - OCaml code from Coq
  - Verified binary code from HOL4 functions via CakeML tools
  - Assembly code from Gallina (Coq)
    - Œuf, CertiCoq
  - Isabelle/HOL
    - ML, OCaml, Haskell, Scala
    - Interface to CakeML
  - C code from PVS (PVS2C)
  - ACL2 is both a logic *and* a programming language (Applicative subset of Common Lisp)

© 2018 Rockwell Collins. All rights reserved.

Related Work: Theorem Provers and Verified/ Verifying Compilers (cont'd.)

- Verified/Verifying compilers for conventional programming languages are a boon for establishing compiler correctness, but do nothing to prove correctness of the source code
- We are particularly interested in tools that provide a verified connection between high-level specifications of algorithms and data structures expressed in a logic (where algorithm correctness proofs can be readily performed) and (hopefully efficient) low-level implementations (the bits in the box)



© 2018 Rockwell Collins. All rights reserved.

The **`sized`** Declarator and Compilation to Array-Based Form

- The DASL **`sized`** declarator informs the toolchain that an otherwise unbounded datatype declaration has limited size:

```
sized pq: PQType (MAX_VERTICES);
```

- sized datatypes can be compiled to an array-based form with destructive updates, similar to the way that ACL2 single-threaded objects (stobjs) are compiled

- Array-based form greatly simplifies code generation for GPUs and hardware

© 2018 Rockwell Collins. All rights reserved.

Example DASL datatype: Binary Search Tree (BST)

- Binary Search Tree from Sedgewick and Wayne's *Algorithms* (4th edition) translated into DASL:

```
package BSTree =

const MAX_VERTICES : uint = 500000;

datatype BSTree
  = Leaf
  | Node :
     (key   : uint,   -- Key (sorted) = 0 => "null key"
      val   : uint,   -- Associated data = 0 => "null val"
      size  : uint,   -- Nodes in this subtree
      left  : BSTree,
      right : BSTree);

sized BSTRoot: BSTree(MAX_VERTICES);
```


© 2018 Rockwell Collins. All rights reserved.

Binary Search Tree (BST) (cont'd)

- BST has operators for isEmpty(), sizeOf(), getVal(), insert(), delete(), deleteMin(), deleteMax(), etc.

- Example BST Operator in DASL: deleteMin()

```
function deleteMin(bst: inout BSTree) {
  match bst {
    'Leaf =>
      skip;
    'Node n =>
     { match n.left {
         'Leaf =>
            bst := n.right;
         'Node nl =>
            deleteMin(n.left);
       }
       n.size := 1 + sizeOf(n.left) + sizeOf(n.right);
     }
  }
}
```


© 2018 Rockwell Collins. All rights reserved.

DASL Graph Datatypes

- Another unique DASL feature is a specialized graph datatype declarator, and its associated sized declarator:

```
graphtype DKGraph (nodeLabel = vertexLabelTy,
                   edgeLabel = edgeLabelTy);

sized dkg: DKGraph (MAX_VERTICES, MAX_EDGES_PER_VERTEX);
```

- The DASL toolchain compiles this declaration to an array-based form, and generates several associated functions for manipulating the array-based form:

```
getOutEdges(), setOutEdges(), addEdge(), labelVertex(),
 labelEdge(),…
```


© 2018 Rockwell Collins. All rights reserved.

Example graphtype: Depth-First Search

- graphtype declaration

```
type vertexLabelTy = uint;
type edgeLabelTy = uint;

graphtype graph (nodeLabel = vertexLabelTy,
                 edgeLabel = edgeLabelTy);

const MAX_NODES : uint = 50000;
const MAX_EDGES : uint = 4;
type vertex = uint;

sized theGraph : graph (MAX_NODES, MAX_EDGES);
```


© 2018 Rockwell Collins. All rights reserved.

Depth-first search in DASL

```
function DFS_span (vtarget: in vertex, G: in graph,
                   spanning_tree: inout BST,
                   fringe: inout vertex_pair_list) {
  var v, vpred: vertex;
  in
    match fringe {
      -- Out of vertex pairs to process
      'Empty => skip;
      'Node n => {
        -- Found target vertex
        if exists(vtarget, spanning_tree) then
          skip;
        else {
          (v, vpred) := n.elt;
          rest(fringe);
          -- if v already found, on to the next fringe element
          if exists(v, spanning_tree) then
            DFS_span(vtarget, G, spanning_tree, vertex_pair_list);
          else {
            mark(v, vpred, spanning_tree);
            explore(MAX_EDGES, v, G, spanning_tree, vertex_pair_list);
            DFS_span(vtarget, G, spanning_tree, vertex_pair_list);
          }}}}}
```

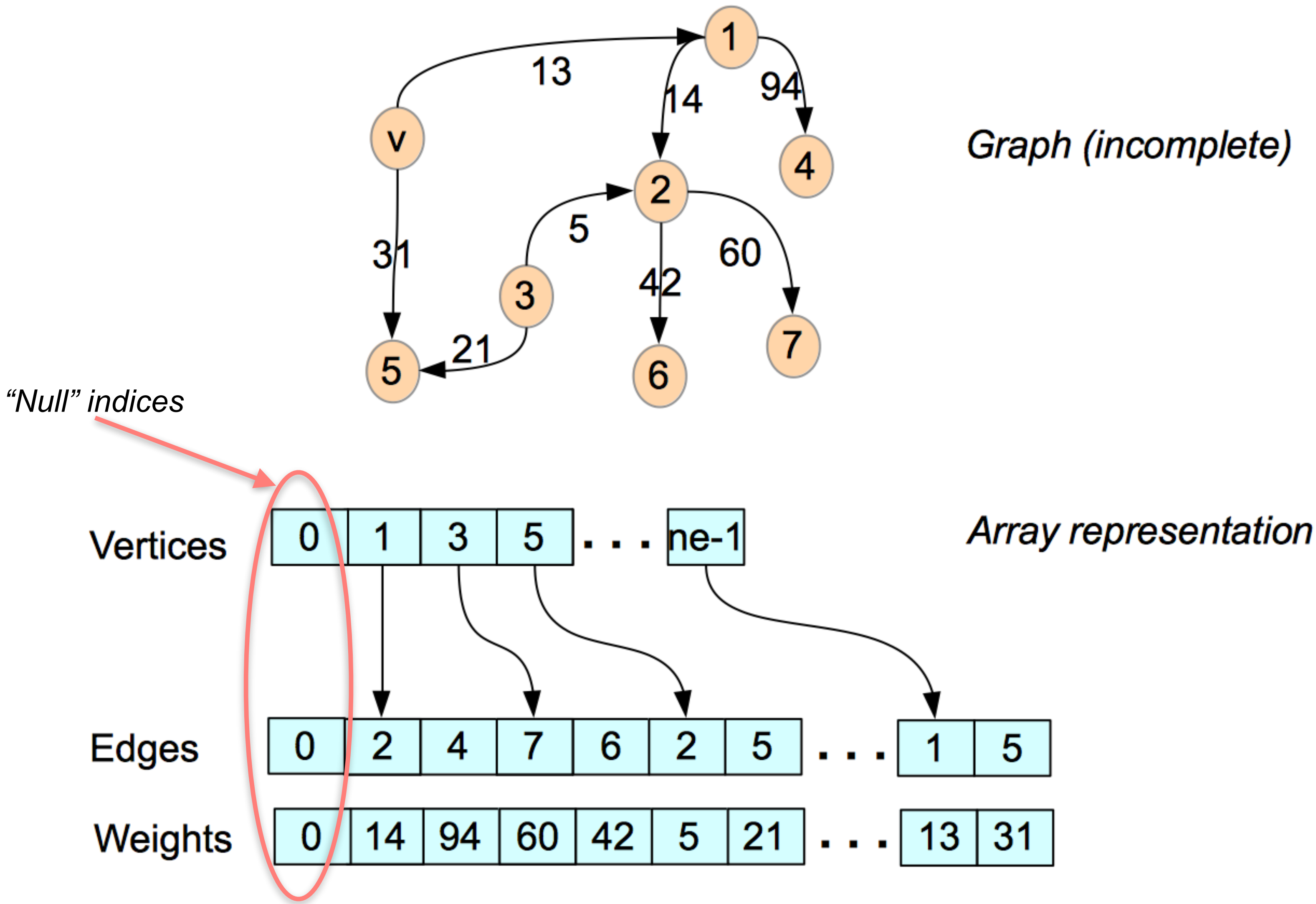

© 2018 Rockwell Collins. All rights reserved.

Array-Based Graph Representation

- Based on a data structure layout approach created for efficient GPU execution (Harish and Narayanan, HiPC 2007); used to code Dijkstra’s All-Pairs Shortest Path algorithm (APSP)
- Amenable to efficient CUDA, OpenCL implementation, as well as hardware implmentation (VHDL)
- Implementated APSP using ACL2 single-threaded object (ACL2 Workshop 2013)
  - Execution of Dijkstra’s shortest path algorithm on compiled graph using stobjs was linear in number of vertices up to at least 1 million vertices at 10 edges per vertex
- DASL compiler analyzes **`datatype`**, **`graphtype`**, and **`sized`** declarations, creates appropriate array-based layout, and instantiates runtime functions

© 2018 Rockwell Collins. All rights reserved.

Graph Compilation Example, Two Edges per Vertex



© 2018 Rockwell Collins. All rights reserved.

Prototyping DASL Data Structure Design in ACL2

Even though we are implementing DASL in HOL4, we chose to prototype the DASL data structure design in ACL2. Why?

- ACL2 is the most capable system we know of for the creation, proof, and execution of formal specifications
  - Using ACL2, we have been able to easily scale our data structure prototypes to millions of vertices and edges
- ACL2 provides sophisticated proof libraries (books) for reasoning about aggregate data structures
- Single-threaded objects (stobjs) provide functional data structure definitions with destructive "under-the-hood" implementations.
- Guards combine a type-like discipline with the power of proof

© 2018 Rockwell Collins. All rights reserved.

Prototyping DASL Data Structure Design in ACL2 (cont'd)

- ACL2 is a mostly-automated theorem prover, and is quite adept at automated inductive proofs
- Tail recursion in ACL2 combines recursive functional style with efficient compilation to loops
- ACL2's simple packaging facility provides separate namespaces for datatypes/graphtypes
- All functions admitted to ACL2 must first be proven to terminate
  - This encourages the ACL2 developer to explicitly consider termination issues when writing functions

- We have thus constructed a Rudimentary ACL2 Semantic Laboratory for DASL:

© 2018 Rockwell Collins. All rights reserved.

Prototyping DASL Data Structure Design in ACL2 (cont'd)

- ACL2 is a mostly-automated theorem prover, and is quite adept at automated inductive proofs
- Tail recursion in ACL2 combines recursive functional style with efficient compilation to loops.
- ACL2's simple packaging facility provides separate namespaces for datatypes/graphtypes
- All functions admitted to ACL2 must first be proven to terminate
  - This encourages the ACL2 developer to explicitly consider termination issues when writing functions

- We have thus constructed a Rudimentary ACL2 Semantic Laboratory for DASL: **RASL DASL**

© 2018 Rockwell Collins. All rights reserved.

RASL DASL

- Source of original prototype of DASL array-based memory layout and DASL datatype runtime API
    - No garbage generation/collection overhead
- Utilizes ACL2 single-threaded objects for all data structures
    - RASL DASL functions are applicative, yet implement array updates destructively “under-the-hood”
    - Each data structure defined in its own package
- All target functions are written in tail-recursive style, so that recursion can be compiled to looping
- Embrace the restriction that all ACL2 functions must be proved to terminate
- All functions have guards for “super-defensive” programming

© 2018 Rockwell Collins. All rights reserved.

Binary Search Tree Declaration in RASL DASL

```
(in-package "BST")

(defconst *MAX_VTX* 65535)
(defconst *MAX_VTX1* (1+ *MAX_VTX*)) ;; 2**16
(defconst *MAX_EDGES_PER_VTX* 2)
(defconst *MAX_EDGE* (* *MAX_VTX* *MAX_EDGES_PER_VTX*))
(defconst *MAX_EDGE1* (1+ (* *MAX_VTX* *MAX_EDGES_PER_VTX*)))
(defconst *MAX_EDGE_MINUS* (1+ (- *MAX_EDGE* *MAX_EDGES_PER_VTX*)))

(defstobj Obj
;; padding -- keeps ACL2 from turning (nth *VTXHD* Obj) into (car Obj)
  (pad :type t :initially 0)
  (vtxHd :type (integer 0 65535) :initially 0)
  (vtxTl :type (integer 0 65535) :initially 0)
  (vtxCount :type (integer 0 65535) :initially 0)
;; (V) This contains a pointer to the edge list for each vertex
  (vtxArr :type (array (integer 0 131069) (*MAX_VTX1*)) :initially 0)
;; (K) Keys for each vertex
  (keyArr :type (array (integer 0 *) (*MAX_VTX1*)) :initially 0)
;; (D) Data Value array
  (valArr :type (array (integer 0 *) (*MAX_VTX1*)) :initially 0)
;; (E) This contains pointers to the vertices that each edge is attached to
  (edgeArr :type (array (integer 0 65535) (*MAX_EDGE1*)) :initially 0)
 :inline t)
```


© 2018 Rockwell Collins. All rights reserved.

Sample BST Function in RASL DASL

```
((defun getVal (count key vtx Obj)
  (declare (xargs :stobjs Obj
                  :guard (and (natp count) (natp key) (natp vtx))))
  (cond
   ((not (mbt (Objp Obj))) 0)    ;; Only positive values stored. 0 = 'null'.
   ((not (mbt (natp count))) 0)
   ((not (mbt (natp key))) 0)
   ((not (mbt (natp vtx))) 0)
   ((zp count) 0)
   ((zp key) 0)                  ;; Only positive keys stored. 0 = 'null'.
   ((zp vtx) 0)
   ((> vtx *MAX_VTX*) 0)
   ((mbe :logic (zp (vtxCount Obj))
         :exec (int= (vtxCount Obj) 0)) 0) ;; no vertices
   ((zp (keyArri vtx Obj)) 0)
   ((< key (keyArri vtx Obj))
    (getVal (1- count) key (edgeArri (left (vtxArri vtx Obj)) Obj) Obj))
   ((> key (keyArri vtx Obj))
    (getVal (1- count) key (edgeArri (right (vtxArri vtx Obj)) Obj) Obj))
   ;; (= key (keyArri vtx Obj))
   (t (valArri vtx Obj))))

(defmacro getV (key Obj)
  `(getVal (vtxCount ,Obj) ,key (vtxHd ,Obj) ,Obj))
```


© 2018 Rockwell Collins. All rights reserved.

Depth-First Search in RASL DASL

fringe: doubly-linked list of (vtx, vtx_prev) pairs

mark/spanning tree (Binary Search Tree)

```
(defun dfs_span (count vtarget gObj BST::Obj DLPR::Obj)
  (declare (xargs :stobjs (gObj BST::Obj DLPR::Obj)
                  :guard (and (natp count) (natp vtarget))))
  (cond
   ((not <recapitulation of :guard conditions>) (mv BST::Obj DLPR::Obj))
   ((zp count) (mv BST::Obj DLPR::Obj))
   ((> vtarget *MAX_VTX*) (mv BST::Obj DLPR::Obj))
   ((BST::existp vtarget BST::Obj) (mv BST::Obj DLPR::Obj)) ;; Found target vtx
   ((zp (DLPR::ln DLPR::Obj)) (mv BST::Obj DLPR::Obj))
   (t (mv-let (v vpred) (DLPR::nthelem 0 DLPR::Obj)
        (cond
         ((not (posp v)) (mv BST::Obj DLPR::Obj))
         ((> v *MAX_VTX*) (mv BST::Obj DLPR::Obj))
         ((not (posp vpred)) (mv BST::Obj DLPR::Obj))
         ((> vpred *MAX_VTX*) (mv BST::Obj DLPR::Obj))
         ((BST::existp v BST::Obj)
          (seq2 BST::Obj DLPR::Obj
                (BST::nop BST::Obj)
                (DLPR::rst DLPR::Obj)
                (dfs_span (1- count) vtarget gObj BST::Obj DLPR::Obj)))
         (t (seq2 BST::Obj DLPR::Obj
                  (mark v vpred BST::Obj)
                  (seq DLPR::Obj
                       (DLPR::rst DLPR::Obj)
                       (explore *MAX_EDGES_PER_VTX* v gObj BST::Obj DLPR::Obj))
                  (dfs_span (1- count) vtarget gObj BST::Obj DLPR::Obj)))))))))
```


© 2018 Rockwell Collins. All rights reserved.

Functional Correctness Proofs in RASL DASL

- Since we have a formalization of the DASL data structure form in ACL2, we should perform functional correctness proofs
- Doubly-linked list correctness proofs are relatively simple
- For Binary Search Trees, we have a nice correctness property, namely that inorder traversal of the BST yields a sorted list:

```
(defthm bstp-ordered-posp-of-inorder-traversal
  (implies
   (bstp Obj)
   (ordered-posp (inorder Obj))))
```

where **bstp** is a BST well-formedness property that says, mainly, that all non-zero keys of the "left" children of any given non-zero vertex **vtx** of the BST are less than (**keyArri vtx Obj**), and that the keys of the "right" children of **vtx** are greater than **(keyArri vtx Obj)**

© 2018 Rockwell Collins. All rights reserved.

Functional Correctness Proofs (cont'd.)

- Thus, for any Binary Search Tree mutator function **`mut`**, we need to prove:

```
(defthm mut-preserves—ordered-posp-of-inorder-traversal
  (implies
   (bstp Obj)
   (ordered-posp (inorder (mut <args> Obj)))
```

which, in turn, requires that **`bstp`** is upheld by **`mut`**. Not surprisingly, this isn't easy using a stobj-based infrastructure, what with arrays, tail-recursive functions, and whatnot

- The hard part is establishing that the "keys of left subtree all less than" and the "keys of right subtree all greater than" functions continue to hold after **`mut`**

© 2018 Rockwell Collins. All rights reserved.

Functional Correctness Proofs (cont'd.)

- To perform the proofs of the **`all-lt-p`** and **`all-gt-p`** functions (not shown for space), we need to define some lower-level well-formedness invariants on the underlying stobj representation of the BST, and show that they hold after **`mut`**.
- Many possible invariants — how do we know which to define?
  - Use "The Method": Try some high-level proofs, and see what low-level properties pop up in the failed subgoals
- Using "The Method" led to three well-formedness predicates for the BST stobj:
  - The non-zero vertex array element for vertex index **`vtx`** is

```
(1+ (* (1- vtx) *MAX_EDGES_PER_VTX*))
```

  - All non-zero edge array elements are unique, i.e. no two edges "point to" the same vertex
  - All non-zero edges "point to" a non-zero entry in the vertex array; no "dangling edges"

© 2018 Rockwell Collins. All rights reserved.

Functional Correctness Proofs (cont'd.)

- The latter two invariant functions are computationally expensive; however, they are only called by functions that are used in proofs, and are not called by any runtime BST function
- Using these discovered wellf-formedness predicates, we have been able to prove that the **`all-lt-p`** and **`all-gt-p`** functions continue to hold after the **`insert`** mutator
  - The "long pole in the tent" to proving the top-level BST functonal correctness properties
- Currently working to make these proofs more efficient by disabling more functions and unnecessary theorems
  - **`all-lt-p`** proof takes approximately 30 minutes of proof time on an ancient 2012 MacBook Pro

© 2018 Rockwell Collins. All rights reserved.

Status and Next Steps

- *Completed*:
  - HOL4 DASL toolchain supports datatypes and graphtypes
  - Basic Data structure prototyping using RASL DASL complete
    - Several data structures developed, e.g. stacks, lists, binary search trees, priority queues, directed graphs
    - Several applications implemented, including breadth/ depth-first search, lexer/parser, and inference engine
- *Next Steps*:
  - Complete binary search tree functional correctness proofs
    - “Hard part” done
  - Improve book certification times
  - Complete DASL proof infrastructure in HOL4
  - Demonstrate end-to-end DASL datatype proof using CakeML
  - Demonstrate DASL compilation to VHDL/FPGA

© 2018 Rockwell Collins. All rights reserved.