Address Enumeration and Reasoning Over Linear Address Spaces

David Greve
Rockwell Collins Advanced Technology Center
Cedar Rapids, IA
dagreve@rockwellcollins.com

10th November 2004

Abstract

At the November 2002 ACL2 users group meeting Rockwell Collins submitted a set of problems concerning reasoning about data structures embedded in a linear address space[2]. These problems are not mere academic exercises. Rather, they are generalizations of the sort of problems that needed to be solved in order to verify formally the intrinsic partitioning mechanism of the AAMP7[1]. In this paper we discuss how address enumeration was used in the AAMP7 proofs to solve the problem of linear address space reasoning.

1 AAMP7 GACC Library

The AAMP7 is a microcoded microprocessor designed for use in embedded systems. The AAMP7 provides a novel architectural feature, intrinsic partitioning, that enables the microprocessor to enforce an explicit communication policy between applications. We used ACL2 to show that the AAMP7 intrinsic partitioning mechanism works as expected. To do this, we created a number of partitioning models, ranging from a high level specification to a low level design model of the AAMP7[6].

The low level design model of the AAMP7 takes a microcoder's view of the system. This is done to ease the labor-intensive process of validating the formal model against the microcode. While the AAMP7 employs well-defined data structures to implement intrinsic partitioning, the data structures are embedded in a linear address space and are available to the microcoder only via primitive read and write operations on memory. For the microcoder, accessing a particular field of a data structure typically means adding an appropriate offset value to the base pointer of the structure and then reading from memory an appropriate number of bytes starting at the computed address. Because our model remains true to the microcoder's view of the system, we are forced to describe and reason about the operations of the microcode as it manipulates these data structures at this level of detail.

There are a variety of techniques that one might use to reason about data structures in a linear address space[4, 5]. We chose to reason about them using address enumeration. In address enumeration, one or more functions are used to crawl through a memory space and construct a list (actually, a bag) of all of the address locations that are used to implement a particular data structure. That list is then used to construct hypotheses (such as disjointness) concerning the data structure of interest. A library was developed to facilitate this kind of reasoning on the AAMP7 project. The library provides a simple language for describing data structures and a methodology for simplifying expressions that manipulate them. We call this library the AAMP7 GACC (**G**eneralized **ACC**essor) library.

1.1 Primitive Operations

The memory of the AAMP7 is specified using the `defrecord[3]` macro.

```
(defrecord memory
  :rd rd
  :wr wr
  :typep wint8
  :fix   wfix8)
```

The `defrecord` macro is a wrapper around the standard records library available in the ACL2 distribution. The `defrecord` macro defines two operations: a read operation (`rd`) for accessing a particular memory location and a write operation (`wr`) that can be used to update the values of memory locations. The macro also provides a number of theorems to aid in reasoning about these primitive operations over memory. The primary advantage of the `defrecord` macro is that the read operation is coerced to return a value of a particular type. In the example above, the function `wfix8` is used to coerce values to `wint8`, which is just a byte (an 8-bit integer).

It is trivial to show the following non-interference property of `rd` and `wr`:

```
(defthm rd-over-wr
  (implies
    (not (equal ra wa))
    (equal (rd ra (wr wa v ram))
           (rd ra ram))))
```

1.2 Multi-Byte Operations

While the AAMP7 employs a byte addressable memory, most of the data structure fields it manipulates are words composed of multiple bytes. To accommodate this, we extend the primitive accessors (`rd/wr`) and define functions (`rx/wx`) that manipulate words of arbitrary[1] size.

In conjunction with the `rx/wx` functions, we also introduce a function that enumerates all of the primitive byte-address locations touched by a given `rx` or `wx` function. This function is called `SBLK` (sized block). Here we show how we use `SBLK` to enumerate all of the atomic memory address locations needed to state the non-interference property for `rx/wx`:

```
(defthm rx-over-wx
  (implies
    (disjoint (sblk wsize wptr)
              (sblk rsize rptr))
    (equal (rx rsize rptr (wx wsize wptr ram))
           (rx rsize rptr ram))))
```

1.3 Data Structures

A relatively simple data structure description language was developed for use in the AAMP7 proofs. The language allows the user to specify, in a consistent manner, all of the data structures used by the AAMP7.

1.3.1 Skels and Specs

A particular data structure can be defined by a `skel` (or skeleton - think of it as the skeleton upon which the data structure is implemented), which includes a *base* pointer to the data structure of interest and a list of slots. The list of slots is called a *spec* (or specification) where each slot represents a particular field of the data structure.

```
(defstructure skel
  base
  spec)
```

A `slot` is defined as follows:

```
(defstructure slot
  name
  off
  size
  val
  type
  skel)
```

[1]The functions `rx` and `wx` accept a size argument specifying the number of bits they return. This value is rounded to the next largest, non-zero multiple of 8 to enable the use of the primitive `rd/wr` operations.

A `slot` describes a particular field of the data structure. The *name* entry is a symbolic name for the field in the data structure. The *off* entry is the offset of the field from the base (or root) pointer for the data structure. The *size* entry indicates how large the value of this field is in bits. The *val* entry is used to store the value of the field. The *type* entry tells us whether this is a pointer field or a data value field. If the slot is of type pointer, then the *skel* field is expected to contain the `skel` representation of the data structure to which this field points. If the slot is not of type pointer, the *skel* field is ignored. Note that the existence of the skel field enables us to create recursive descriptions of recursive data structures.

An example spec for a recursive schedule data structure with 3 fields (Next, Time_Count, and Saved_State) might appear as follows:

```
(defun schedule-spec (n)
  (list
    (slot :Next #x00 32 0 :ptr
          (skel 0 (schedule-spec (1- n))))
    (slot :Time_Count #x04 32 0 :int (skel 0 nil))
    (slot :Saved_State #x08 32 0 :ptr
          (skel 0 (state-spec))))
```

1.3.2 Reading and Writing using specs

We can use a spec to read information out of a memory or to write information into a memory. To read information from a memory, we use the `g*-spec` (g for get) function. This function takes an op argument[2], a pointer to the base of the data structure of interest, a spec for the data structure, and a memory. From those arguments, `g*-spec` returns a spec whose value fields have been updated as governed by the op argument.

```
(g*-spec op ptr spec ram) -> spec
```

A partial example of the behavior of `g*-spec` follows:

```
(g*-spec op ptr (cons slot spec) ram) =
  (let ((skel (slot-skel slot))
        (size (slot-size slot))           ; size
        (addr (+ ptr (slot-off slot))))   ; address
    (let ((val   (rx size addr ram))          ; read
          (gspec (g*-spec op (skel-base skel) ; recurse
                             (skel-spec skel)
                              ram)))
     (cons (update-slot slot
             :val  val
```

[2]The *op* argument is used to tune the behavior of the various *-spec functions. For example, the op argument can be used to dictate which types of fields from the spec to operate on: pointer fields, value fields, or both.

```
                     :skel (update-skel skel :spec gspec))
                   (g*-spec op ptr spec ram)))
```

The function `s*-spec` (s for set) allows us to write information from a spec into memory. The `s*-spec` function takes an op argument, a pointer to the base of the data structure of interest, a spec for the data structure, and a memory. From those arguments, `s*-spec` returns a memory that has been updated to reflect the values stored in the spec.

```
(s*-spec op ptr spec ram) -> ram
```

A partial example of the behavior of `s*-spec` follows:

```
(s*-spec op ptr (cons slot spec) ram) =
  (let ((skel (slot-skel slot))
        (size (slot-size slot))
        (addr (+ ptr (slot-off slot)))
        (val  (slot-val  slot)))
    (let ((ram (wx size addr val ram))) ; write
      (let ((ram (s*-spec op (skel-base skel) ; recurse
                             (skel-spec skel)
                             ram)))
        (s*-spec op ptr spec ram))))
```

Note that `g*-spec` and `s*-spec` play roles similar in nature to `rx` and `wx` with respect to memory except that `g*-spec` and `s*-spec` operate on entire data structures, rather than on individual values.

One other function useful in reasoning about the behavior of `g*-spec` and `s*-spec` is `f*-spec` (f for flatten). The function `f*-spec` takes an op argument, a pointer, and a data structure spec and returns a bag containing all of the primitive byte-address locations that would be touched by a `g*-spec` or `s*-spec` function given those same arguments[3].

```
(f*-spec op ptr spec) -> list
```

A partial example of the behavior of `f*-spec` follows:

```
(f*-spec op ptr (cons slot spec)) =
  (let ((skel (slot-skel slot))
        (addr (+ ptr (slot-off slot)))
        (size (slot-size slot)))
    (append (list (sblk size addr))
            (f*-spec op (skel-base skel)
                        (skel-spec skel))
            (f*-spec op ptr spec))))
```

[3]To be precise, `f*-spec` returns a bag of bags of such addresses; the desired bag is obtained using (`flat (f*-spec ..)`)

Here we use f*-spec to enumerate the addresses used by the two data structures of interest and, ultimately, to state the non-interference theorem for g*-spec and s*-spec:

```
(defthm g*-spec-over-s*-spec
  (implies
    (disjoint (flat (f*-spec gop gptr gspec))
              (flat (f*-spec sop sptr sspec)))
    (equal (g*-spec gop gptr gspec
                    (s*-spec sop sptr sspec ram))
           (g*-spec gop gptr gspec ram))))
```

This theorem states that the addresses composing the data structure specified by (gptr,gspec) are disjoint from those composing (sptr,sspec) and, as a result, modifications of the second data structure (manipulated by s*-spec) do not change the values stored in the first data structure (accessed by g*-spec).

The use of specs and a relatively small set of associated functions (g*-spec, s*-spec, f*-spec, etc) enables us to specify the various data structures employed by the AAMP7 microcode and to verify the correctness of the AAMP7 intrinsic partitioning mechanism.

2 Reasoning Techniques

The AAMP7 GACC library provides the ability to extract a data structure representation from memory (using g*-spec) and the ability to enumerate all of the memory addresses used by that structure (using f*-spec). The next step is to enable efficient reasoning about these enumerations. We found that the most appropriate data structure for representing and reasoning about these enumerations were bags (multi-sets). The bag relationships most commonly used to describe the properties of addresses and bags of addresses are perm (permutation), memberp, subbagp, disjoint, and unique.

Unfortunately, the standard approach of simply expanding and normalizing terms expressing such relationships results in logical expressions whose size is quadratic in the number of interesting elements. Another difficulty with reasoning about data structures is that establishing certain relationships may require existential reasoning (i.e., free variables in the hypothesis). Both of these issues plagued the AAMP7 verification effort. In light of these issues, two ACL2 capabilities that proved crucial to the successful deployment of the AAMP7 GACC library were meta rules and bind-free[7].

2.1 Meta Rules

Because the uniqueness predicate compares each element of the bag with every other element of the bag, a straighforward simplification of a uniqueness statement produces a quadratic number of sub terms. For small examples this is acceptable, but it becomes more of an issue as the size of the data structures being reasoned about increases. Our solution to this problem is to avoid expanding such terms and, instead, to employ meta rules. Meta rules are rules that extend the ACL2 simplifier with user-provided functions to perform syntactic simplification of specified terms. This meta rule simplification is proven sound for some specific set of functions, upon which the rule can then be programmed to fire.

For the AAMP7 program we developed meta rules that are able to simplify expressions involving `subbagp`, `memberp` and a handful of other related functions. For example, it is obvious from inspection that the following is true:

```
(memberp c (list a b c w x y))
```

To simplify this expression, the meta evaluator simply searches the list for an occurrence of the symbol c. If it finds one, it signals success and rewrites the expression to true. The use of such syntactic evaluation enables us to turn off the definitions of all of the functions of interest (`memberp`, `subbagp`, `disjoint`, `unique`) and to use meta rules to simplify such expressions.

2.2 Bind-Free

In order to reason about data structures we must establish the independence (non-interference) of the various functions that operate on those data structures. The most pressing questions are usually of the form `(disjoint x y)`. How does one establish (non-trivial) disjointness? One possible rule is the following:

```
(defthm disjoint-from-disjoint
  (implies
    (and
      (disjoint a b)
      (subbagp x a)
      (subbagp y b))
    (disjoint x y)))
```

This rule says that if there exists an `a` and `b` such that `a` and `b` are disjoint and one can show that `x` and `y` are subbags of `a` and `b`, then one can conclude that `x` and `y` are disjoint. Another rule might be:

```
(defthm disjoint-from-unique
  (implies
    (and
      (unique bag)
      (unique-subbagsp x y bag))
    (disjoint x y)))
```

This rule says that, if there exists a unique bag, `bag`, and one can establish that `x` and `y` are both (unique) subbags of `bag`, then one can conclude that `x` and `y` are disjoint.

What is interesting about these rules is that they both require existential reasoning: they say "if there exists some arbitrary bag or pair a,b such that ..". Using such a rule requires ACL2 to perform free variable matching. Unfortunately, for the sake of completeness we must include a number of other, similar cases, with `x` and `y` separated from either `bag` or `a` and `b` by varying numbers of subbag relationships. By the time we are done, we have accumulated a set of very expensive rules that will be evaluated many thousands of times for reasoning about data structure operations of even moderate complexity.

To solve this problem, the AAMP7 GACC library employs ACL2's bind-free facility. Rather than have a large set of rules for establishing disjointness, the GACC library provides a single rule that searches the hypotheses for a suitable argument for why `x` and `y` might be disjoint. It employs heuristics suggested by the two rules above, and is able to resolve abitrarily deep nestings of relationships of this form. Once it discovers a suitable argument for why the two values are disjoint, it binds all of the relevant formulae to the arguments of the disjointness hypothesis. ACL2 then expands the hypothesis and employs simple reasoning techniques, including meta rules, to satisfy the resulting expression.

```
(defthm disjoint-computation
  (implies
    (and
      (bind-free
        (bind-disjoint-argument
           nil 'key 'xlist 'x0
           'ylist 'y0 'z 'zlist
           'z0 x y acl2::mfc acl2::state)
        (key xlist x0 ylist y0 z zlist z0))
      (disjoint-hyp
        key x xlist x0 y ylist y0 z zlist z0))
   (disjoint x y))
```

The `disjoint-hyp` term reflects the logical reasoning employed by `bind-disjoint--argument`. Without further explaination the body of `disjoint-hyp` has the following form, wherein the two `key` cases correspond to generalizations of the `disjoint-from-disjoint` and `disjoint-from-unique` lemmas:

```
(defun disjoint-hyp (key x xlist x0 y ylist
                     y0 z-syn zlist-p z0-q)
  (cond
    ((equal key ':disjoint)
      (and (subset-pair x0 y0 zlist-p z0-q)
           (hide-disjoint zlist-p z0-q)
           (subset-chain x xlist x0)
           (subset-chain y ylist y0)))
    ((equal key ':unique)
      (and (unique-subset-chain
              x0 y0 z-syn zlist-p z0-q)
```

```
            (subset-chain x xlist x0)
            (subset-chain y ylist y0)))
       (t nil)))
```

3 Address Enumeration and Reflexive Functions

In his paper[5], J Moore describes a memory tagging technique for reasoning about data structures in a linear address space. In memory tagging, a data structure is maintained that describes, for each memory location of interest, how that location will be interpreted by functions of interest. This information is then used to prove the essential non-interference theorems.

The AAMP7 GACC library, on the other hand, uses *address enumeration* for describing data structures. Hanbing Liu's solution to the Rockwell challenge, for example, employed address enumeration techniques[4]. Hanbing used address enumeration to define equivalence relationships over memories and to re-factor functions over memory to separate data structure traversal from data structure modification.

An early concern with the address enumeration technique was that it was not sufficiently general to enable non-interference proofs about reflexive (multiply recursive) functions. An example reflexive function might be `mark-raw`, shown here:

```
(defun mark-raw (n ptr ram)
  (if (zp n) ram
    (let ((p1 (g (+ ptr 1) ram))
          (v2 (g (+ ptr 2) ram))
          (v3 (g (+ ptr 3) ram))
          (p4 (g (+ ptr 4) ram)))
     (let ((ram (s (+ ptr 3) (+ v2 v3) ram)))
      (let ((ram (mark-raw (1- n) p1 ram)))
       (mark-raw (1- n) p4 ram))))))
```

Our various attempts to reason about such functions met with frustration as we found ourselves needing to use the very rule we were trying to prove as we attempted to induct over the reflexive function. While we identified this concern early in the AAMP7 program, we did not anticipate it being of practical importance in the AAMP7 proofs since the intrinsic partitioning mechanism did not involve the use of such functions. Nonetheless, our early failures in trying to reason about reflexive functions threatened the general applicability of the address enumeration approach.

The solution, it turns out, is to use "access guards" to simplify and generalize reasoning about reflexive functions. Access guards play a role similar to what we find in recursion guards. A recursion guard is an additional, natural number argument added to the signature of a recursive function. The recursion guard is checked in each recursive call. If the guard is zero, the function terminates. If not, the body of the function is applied to the arguments and any recursive calls decrement the recursion guard. The recursion guard provides a short-cut for admitting functions with otherwise arbitrarily complex measures. One may then chose (or be driven) to formulate the arbitrarily complex measure funcation and use it to prove the equivalence of the guarded function and a version of the same function without the recursion guard.

An access guard, likewise, is an additional argument added to the signature of a function. At the beginning of the function the access guard is checked against all of the memory transactions that may be performed by the body of the function. If any of the transactions are not allowed according to the access guard, the function returns an unmodified memory. If all of the transactions are allowed, the body of the function is applied to its arguments and the access guard is passed on to any recursive calls. Here is a version of `mark-raw` with an access guard:

```
(defun mark-body-defs (ptr)
  (list (+ ptr 3)))
;
(defun mark-guard (defs n ptr ram)
  (if (subbagp (mark-body-defs ptr) defs)
   (if (zp n) ram
    (let ((p1 (g (+ ptr 1) ram))
          (v2 (g (+ ptr 2) ram))
          (v3 (g (+ ptr 3) ram))
          (p4 (g (+ ptr 4) ram)))
     (let ((ram (s (+ ptr 3) (+ v2 v3) ram)))
      (let ((ram (mark-guard defs (1- n) p1 ram)))
        (mark-guard defs (1- n) p4 ram)))))
   ram))
```

The access guard thus greatly simplifies proofs of non-interference, allowing read over write and commuting theorems to be proven relative to the access guard.

```
(defthm rd-over-mark-guard
  (implies
   (not (member a defs))
   (equal (g a (mark-guard defs n ptr ram))
          (g a ram))))
```

In this way, the access guard decouples the access protection question from the behavior of the function of interest. This decoupling provides precisely the generality we need to prove our fundamental non-interference theorems for reflexive functions. With these theorems in hand, we are then able to instantiate the generic access guard with a function that computes the actual access requirements of the function in question and show that the instantiated, guarded function reduces to a version of the same function without the access guard.

4 Conclusion

We have described how address enumeration techniques have been used in the AAMP7 proofs to solve the problem of linear address space reasoning. The use of specs as a data structure description language enables us to write clear, concise descriptions of the AAMP7 data structures, the *-spec functions enables us to use those descriptions to help characterize the AAMP7 intrinsic partitioning functionality, and the meta and bind-free rules help to automate the verification of AAMP7 system correctness. We have also assuaged our concerns regarding the applicability of address enumeration techniques to the problem of reasoning about reflexive functions.

Future efforts will focus on improving the general applicability of the GACC library. The AAMP7 GACC library was developed in an ad-hoc manner to solve a specific problem and there are a number of design limitations that should be addressed. Our ultimate objective, however, is the development of a general purpose GACC library that enables efficient reasoning about a wide variety of data structures. There is, however, almost always a trade-off between utility (efficiency) and generality (scope). An indication of the utility of a library is its successful application in solving a large problem. A measure of the generality of a library is in solving a large number of small problems. It is our hope that a future GACC library succeeds on both counts.

References

[1] David Greve, Raymond Richards, and Matthew Wilding. A Summary of Intrinsic Partitioning Verification. In *ACL2 Workshop 2004*, November 2004.

[2] David Greve and Matt Wilding. Dynamic Datastructures in ACL2: A Challenge. Technical report, November 2002. `http://hokiepokie.org/docs`.

[3] David Greve and Matthew Wilding. Typed ACL2 Records. In *ACL2 Workshop 2003*, June 2003.

[4] Hanbing Liu. A Solution to the Rockwell Challenge. In *ACL2 Workshop 2003*, June 2003.

[5] J Moore. Memory Taggings and Dynamic Data Structures. In *ACL2 Workshop 2003*, June 2003.

[6] Raymond Richards, David Greve, Matthew Wilding, and W. Mark Vanfleet. The Common Criteria, Formal Methods and ACL2. In *ACL2 Workshop 2004*, November 2004.

[7] Eric Smith, Serita Nelesen, David Greve, Matthew Wilding, and Raymond Richards. An ACL2 library for Bags (Multisets). In *ACL2 Workshop 2004*, November 2004.