Overview of KAS, applications, and proving a clause processor

ACL2 Weekly Seminar - ACES 3.116

4PM, January 27th, 2010

Rob Sumners

Advanced Micro Devices, Inc.

robert.sumners@amd.com

⌊ Outline ⌋

- Overview of KAS
  - Motivation, design, features
- Applications
  - Procedures, heuristics, examples
- Untrusted Clause Processing
  - Are we there yet? No.

⌊ Motivation - 1 ⌋

• Most theorems are proven by simplification or induction followed by simplification

• Most proofs about hardware/software systems reduce to defining and proving invariants

– Proving (inductive) invariants requires considerable case analysis

• Main Idea:

– Define an efficient term rewriter and implement procedures and heuristics as sets of rewrite rules

• Eventually, we want an untrusted clause processor

– So, write KAS in ACL2 and prove it correct - easy.

⌊ Start from Simple ⌋

- Mutually recursive function clique:
  - `(apply-rule trm rl ctx)` – apply a rewrite rule
  - `(try-rules trm rls ctx)` – apply a list of rules
  - `(rewrite-if args ctx)` – rewrite args of `if` term
  - `(rewrite-list lst ctx)` – rewrite a list of args
  - `(rewrite-args args fn ctx)` – rewrite args of a term
  - `(rewrite-step trm ctx)` – rewrite args then apply rules
  - `(rewrite-term trm ctx)` – fixpoint of `rewrite-step`

- Top-level function:

```
(defun simple-rewrite (trm) (rewrite-term trm ()))
```

- `ctx` is a list of equalities which are currently assumed

  – Extended when the true and false branch of an `if` term are rewritten

⌊ KAS Architecture Overview ⌋

- KAS stands for Kernel Architecture Simplifier

  – KAS is best viewed as an optimized elaboration of this simple rewriter

  – Similar to ACL2, KAS uses inside-out, ordered, conditional, rewriting

- How is this a simplifier?

  – Implement simplification on top of KAS as instances of a meta-process

  – Transform terms (soundly) via rewrite rules

  – Support efficient complex user functions to guide application of these rewrite rules

- Interfaces with ACL2 as a trusted clause processor

  – Loads proven rules and definitions from ACL2 world

⌊ Two main areas of optimization ⌋

- Terms and Memory management
  - How do we represent and store terms efficiently?
  - How do we manage this memory?
- Memoization and Context management
  - How do we cache previous computations?
  - How do we deal with changing contexts?

⌊ Terms and Memory management - 1 ⌋

- Terms are main construct manipulated in KAS and ACL2
  - Use large fixnum arrays in stobjs to store nodes in terms
  - Fixnum indexes into these arrays used as pointers
- Many benefits compared to using `cons`, but
  - It is less elegant – mitigated by use of macros
    - Functions and macros also used for print/debug
  - Need for garbage collection – mitigated by node promotion scheme

⌊ Terms and Memory management - 2 ⌋

- Node Promotion Rules

- All nodes are initially "junk" and promoted if one of the following applies:

  - (a) node is a quoted constant or variable

  - (b) node is in normal form in the current context

  - (c) arguments are promoted and matches previous transient node

    - Use simple cache to store previous viable matches

    - Incrementally grow set of promoted nodes – with some user control

⌊ Terms and Memory management - 3 ⌋

- Transient nodes
  - are not uniquely constructed
  - have minimal storage per node
  - are reclaimed efficiently by "stack" deallocation
- Promoted nodes
  - are constructed uniquely
  - include storage for memoized computations
  - are never reclaimed and never demoted

⌊ Memoization and Context management - 1 ⌋

- Need to cache rewrite results to avoid repeated computation

  – Every promoted node includes a *repnode* field pointing to another node

  – An invariant of KAS execution is that a node is always equivalent to its repnode assuming the current context

    ◦ When an equality is assumed from `if` test, a *repnode* is created

- When KAS rewrites a node, it first consults *repnode* as replacement

  – *repnode*s are updated to resulting normal-forms when rewriting completes

- Obviously we need a system for undoing *repnode* assignments when we pop contexts

⌊ Memoization and Context management - 2 ⌋

- Every *repnode* is tagged with a context vector

  – A context vector is a subset of the current context encoded as a bitvector

  – Invariant is every node is equivalent to its *repnode* assuming its context vector

- Every function in main rewrite loop returns context vector along with rewrite result

- An example to demonstrate context management of *repnodes*:

```
(if (= a b) (if (= b c) (= (f a) (f c))
                        (= (f a) (f b)))
            (= (f a) (f a)))
```

- *repnode* is updated or undone for `(f a)` to match equality in each leaf

⌊ Several Additional Optimizations ⌋

- Avoiding Lisp Execution Overhead

  – fixnums, stobjs, and more fixnums – no consing in main loop

  – inlining and tail recursion to avoid overhead of function calls

- Specialized Data Structures

  – *undo stack* which is a stack of lists of "undos" to be performed when popping the context

- Additional Memoization

  – KAS tags nodes which have been rewritten

⌊ User Control and Interfacing - 1 ⌋

• KAS imports conditional rewrite rules proven as ACL2 theorems

• Fine-grained rewrite control supported through `sieve` operator

– Sieves can access ACL2 state and KAS logic stobj

– Sieves can access and update user stobj

– Sieves can determine if a rule is applied or not

– Sieves return a list of updates to the KAS logic stobj

◦ Updates are restricted to have no effect on soundness of KAS

⌊ User Control and Interfacing - 2 ⌋

- The current list of sieve function updates:

operation	side effect
set-var-bound	bind a free variable in a rewrite
set-rule-sieves	modify the filters attached to a rule
set-rule-enabled	enable or disable a rewrite rule
set-rule-ctr	modify counter for number of rule apps
set-node-step	set node allocation incremental step
set-node-limit	set node allocation limits
change-rule-order	change the order of rewrite rules
set-rule-traced	enable or disable rule trace output
set-user-mark	set or clear a boolean mark on a node

⌊ Implemented, Tested, Rescinded ⌋

- Infinite Rewriting
  - Only support unconditional rewriting
  - Not enough benefit for complications in contexts
- Targeted rewriting
  - Only rewrite the subterms which change in a context
  - Required maintaining backpointers
- Allow user to rewrite everything
  - Use special operators for contexts, hypothesis, etc.
- Contextual rewriting
  - Fairly easy to avoid and it complicated proof effort

⌊ Simulating ACL2 simplification ⌋

- Type prescription and Forward chaining

  – Contextual memoization will retain computed facts needed to relieve hypothesis of rules

- Congruence rewriting

  – KAS only supports `equal`, but other equivalences can be "mapped" to `equal` through normalizing functions

```
(defthm set=-to-equal (equal (set= x y) (= (n-s x) (n-s y))))
```

- Linear Arithmetic – a bit more involved

  – Define rules to normalize linear terms placing operands in term order

  – Define rule to selectively combine and factor out linear terms with matching first operands

  – Can be used for other "linear" operators such as set operations

- Other examples: BDDs, Lambda rewriting (mostly), ...

⌊ Example: Case Splitting ⌋

• Introduce identity functions used as stages in meta-process

```
(defun prv (x) x) (defun prv2 (x) x) (defun prv3 (x) x)
```

• Prove rewrite rules to sequence term transitions in meta-process

– Use `sieve`s to define complex functions or functions outside of term transformation

– `case-split` selects a term based on weighted occurrence in `if` tests

```
(defthm (equal (prv3 t) t))

(defthm (equal (prv2 (if x y z)) (if x y z)))
(defthm (equal (prv2 (if x t (hide z))) (if x t (prv z)))
(defthm (equal (prv2 (if x (prv3 y) z))
               (prv2 (if x (prv y) z))))

(defthm (equal (prv x) x))
(defthm (implies (sieve (case-split C))
                 (equal (prv x)
                        (prv2 (if C (prv3 x) (hide x))))))
```

⌊ Example: Failure Reporting ⌋

• A different meta-process for reporting a failing case as a list of predicates

– Designed to work with case splitting process using `rfl` and `gfl` identity functions

```
(defthm (implies (sieve (report-to-cw leaf))
                 (equal (rfl leaf x) x)))
(defthm (implies (sieve (report-to-cw tst))
                 (equal (rfl (if tst tbr fbr) x)
                        (rfl tbr x))))
(defthm (implies (and (sieve (non-nilp tbr))
                      (sieve (report-to-cw (not tst))))
                 (equal (rfl (if tst tbr fbr) x)
                        (rfl fbr x))))

(defthm (equal (gfl x) (fail (rfl x x))))
(defthm (equal (gfl t) t))
```

• Standard **`defthmk`** macro takes a term α and creates a call to KAS with **`(gfl (prv`** α**`))`**

⌊ Application: Proving Invariants - 1 ⌋

• Start with a stuttering refinement for a simple pipeline model

– Stuttering refinement between `ma` level and `isa` level

– Example modified from DLX pipeline by Manolios,Srinivasan

• Predicate defining a matched `ma` state:

```
(defun ma-matches-isa (x)
  (if (commit x)
      (equal (rep (ma x)) (isa (rep x)))
    (and (equal (rep (ma x)) (rep x))
         (< (rank (ma x)) (rank x)))))
```

– `rep` maps `ma` state to `isa` state

– `rank` is well-founded measure on `ma` states

– `commit` defines when `ma` will make `isa` visible step

⌊ Application: Proving Invariants - 2 ⌋

• Idea from Manolios,Srinivasan: let the `ma` steps build invariant

– Leads to brutal case explosion in a few steps

```
(defun maX4 (m) (ma (ma (ma (ma (flush m))))))
(defun maX5 (m) (ma (maX4 m)))
(defun maX6 (m) (ma (maX5 m)))
(defun maX7 (m) (ma (maX6 m)))
(defun maX8 (m) (ma (maX7 m)))

(defthmk maX4-proof (ma-matches-isa (maX4 m)))
(defthmk maX5-proof (ma-matches-isa (maX5 m)))
(defthmk maX6-proof (ma-matches-isa (maX6 m)))
(defthmk maX7-proof (ma-matches-isa (maX7 m)))
(defthmk maX8-proof (ma-matches-isa (maX8 m)))
```

• ACL2 blows up on `maX5`, KAS takes a minute for `maX8`, but how about proof from arbitrary state:

```
(defthmk ma-proof (ma-matches-isa (ma (ma (ma (ma x))))))
```

⌊ Application: Proving Invariants - 3 ⌋

• Extend this basic idea and include finite state search

– Assume domain of `rep` is finite list of booleans

◦ Introduce Skolem constants to model arbitrary data, commands, identifiers, etc.

– Define `next*` as the *run to commit* function:

```
(defun next* (s)
  (declare (xargs :measure (rank s)))
  (if (commit s) (next s) (next* (next s))))
```

• Explore the states starting with `(rep (init))` and transitioning with `(rep (next* s))`

– Use KAS to rewrite `(rep (next* s))` with heuristics to control expansion of `next` function integrated with case splitting

– Iterate until you reach a fixpoint or find a state which invalidates the invarint `(inv (rep s))` which you are trying to prove

⌊ Proving Clause Processor - 1 ⌋

• We would like to prove a `clause-processor` rule for `kernel-simplify`

```
(defun kernel-simplify (cl hint state ls us)
  (declare (xargs :stobjs state ls us))
  (mv-let (erp term ttree state ls us)
      (kern-simplify-main cl hint ls us state)
    (declare (ignore ttree))
    (mv erp (list (list term)) state ls us)))

(defthm correctness-of-kernel-simplify
  (implies (and (pseudo-term-listp cl)
                (alistp a)
     (my-evl
       (conjoin-clauses
         (clauses-result (kernel-simplify cl hint st ls us))) a)))
    (my-evl (disjoin cl) a)))
```

• Where `my-evl` is an evaluator over a useful set of functions for a given problem

– If your application of KAS requires new functions, then a new evaluator will also be needed

• But, we need an invariant to persist on logic state stobj `ls`, so no dice...

⌊ Proving Clause Processor - 2 ⌋

• Ok, we can create local-stobjs and ensure the proper invariants are maintained in all calls

```
(defthm correctness-of-kernel-simplify
  (implies (and (pseudo-term-listp cl)
                (alistp a)
     (my-evl
       (conjoin-clauses
         (clauses-result (kernel-simplify cl hint st))) a)))
    (my-evl (disjoin cl) a)))
```

• Another potential problem. KAS uses theorems from the current world and in order to prove this result...

– ...we will need theorems relating these theorems to `my-evl` computation

⌊ Proving Clause Processor - 3 ⌋

• Fear not, we can generate these theorems and push them through ACL2 (ugly as it may be):

```
(defthm foo (equal <lhs> <rhs>))

... generates ...

(defun-sk my-eqv (x y)
  (forall (a) (equal (my-evl x a) (my-evl y a))))

(defthm foo-my-evl (my-eqv (quote <lhs>) (quote <rhs>)))
```

• And, all we need is to now prove is:

```
(defthm kas-good (my-eqv (kas-rewrite trm state) trm))
```

• No problem....

⌊ Proving KAS sound ⌋

- Overview
  - Proving termination
  - Defining intermediate "models" of KAS
  - Proving equivalence between levels of definition
  - Verifying guards – yeeesh!

- For each step, necessary invariants will need to be defined and shown to be preserved

- But before we can prove anything, we have to deal with some issues first...

⌊ Macro-expansion ⌋

- First problem... expanding macros

  – The definition of the main functions in KAS use a LOT of nested macros

  – Macros are used to generate type declarations, inline function calls, data structures, assertions, ...

- In particular, for inlined function calls, I would like to:

  – Use partial functions and show that the "expanded" body is equal to unexpanded body

  – But, for now, I just cheat and use a different version of the macros to avoid some of the expansion

⌊ More Complications... ⌋

- Part of KAS is the execution of user-defined sieve functions

  – These sieve functions can produce side effects on the logic state

    ◦ Side effects are limited in order to ensure no impact on soundness

  – Define a "generic" version of KAS with an encapsulated user-sieve function

    ◦ Unfortunately, while side effects will not effect soundness, they can impact equivalence

- Intrinsic limitation: KAS reduces ground terms using evaluation

  – Need an axiom equating `ev-fncall-w` with `my-evl` if function defined in `my-evl`

    ◦ Well, we actually need `ev-fncall-w` in the logic first

⌊ Proving KAS terminates ⌋

• Well, rewriting in general is not guaranteed to terminate...

– Not a problem for soundness, since we can just keep a maximum rewrite counter which will decrease with every rule application

◦ Ok, not that easy, because we don't want this counter to decrease when rewrite subterms, so, a lexical pair of `node` and `clock`

• But, we also have a lot of recursion through "data structures"

– This would require carrying along a significant invariant for the stobjs which would be a significant headache for termination proofs, except...

– I purposely designed all traversals to have strictly decreasing values for all "pointers"

◦ Guarded by an efficient check which will throw an error dynamically if the pointers do not satisfy requirement

⌊ Defining Intermediate Models - 1 ⌋

• Incrementally reduce KAS definition to a simpler rewriter which we could prove easily to be sound

• Possible "steps" in reduction:

  – Un-inline inlined function calls

  – Map packed data structures to lists stored in `ls`

  – Map pointer indexes into lists (primarily transform nodes to terms)

    ◦ Here, we also deal with replacing `node=` with `equal`

    ◦ We also have to prove we safely reclaim "junk" nodes

  – Map bit-packed data into lists (mainly sets)

• This ends the proof of the translation from "C" to LISP

⌊ Defining Intermediate Models - 2 ⌋

• We now could continue the process at algorithmic level:

– Remove memo-table

– Remove undo-stack

– Remove contextual memoization

– ... and so on ...

◦ We also now must switch from proving `equal` to proving `my-eqv`

• Well, I decided to stop defining intermediate models once I finished C-to-LISP translation

– The C-to-LISP steps will be automatically defined, algorithmic intermediate models would need to be maintained

⌊ Guard Verification? ⌋

• Honestly, I have not even started on this one..

• Main KAS functions make liberal use of type declarations (mainly `fixnum` declares)

– My hope is that most of the invariant proof work in C-to-LISP translation steps will help discharge most of the guard proof obligations

– But, this is mainly a hope, because I am fully aware that the "edge" cases which come up in proving `fixnum` guards can easily turn into significant invariant definition and proof obligations

• When I get to proving guards, I will probably make modify the definition of some KAS functions to try and "localize" the guard proofs as much as possible

⌊ Conclusions and Questions/Discussion ⌋

- "Development" of main KAS routines has stabilized
  - I have run out of ideas on how to further mangle the code
  - Still working on applications
- I have completed about 60 percent of the soundness proof work
- Questions?
  - First question: Rob, did you find any bugs?
- Final comment: this is still fun!