User Control and Direction of a More Efficient Simplifier in ACL2

ACL2 Workshop 2009

May 11th, 2009

Rob Sumners

Advanced Micro Devices, Inc.
robert.sumners@amd.com

[[[supporting materials now, full release soon enough]]]

⌊ ACL2 Waterfall Overview ⌋

- Preprocess – early simplification
- **Simplify**
- Setup For Induction
  - Eliminate-Destructors
  - Fertilize
  - Generalize
  - Eliminate-Irrelevance
- Induct

⌊ The Need for Simplification ⌋

• The ACL2 simplifier is the primary component of ACL2 theorem proving application

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

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

– Proving (inductive) invariants requires considerable case analysis

– While ACL2 simplification is useful for many proofs, ACL2 simplification is not optimized for efficient case analysis

⌊ Start from A Simple Rewriter ⌋

- 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 ()))
```

- The context `ctx` argument 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

⌊ Memoization and Context management - 3 ⌋

- The question of reducing contexts
  - How to deal with rewriting the following term:

$\alpha \doteq$ `(if (= (f a) (f b)) t (if (= a b) nil t))`

  - 1. Enable context reduction in KAS
    - KAS will rewrite `(not (= (f a) (f b)))` assuming `(= a b)` to determine an invalid context
  - 2. Use case splitting or other rewriting technique
    - rewrite α to be `(if (= a b)` α α`)`

⌊ 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 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

⌊ 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: Pipeline Verification - 1 ⌋

- Prove 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: Pipeline Verification - 2 ⌋

• Neat 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 few minutes for `maX8`, but how about proof from arbitrary state:

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

⌊ 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.

⌊ Current and Future Work ⌋

- Putting together full KAS and library release – with comments

- Continuing effort to integrate invariant discovery tool of Ray,Sumners into KAS

- Mechanical Proof of Soundness for KAS

  – Proven simple rewriter is sound for a fixed evaluator with assumed rules from world

  – Define an architectural definition of KAS and prove it equivalent to simple rewriter

    ◦ Requires an involved invariant definition

  – Prove KAS implementation is equivalent to architectural model

    ◦ Requires much more involved invariant definition