a ”new” ”simplifier”

Rob Sumners

robert.sumners@amd.com

March 19th, 2003

March 26th, 2003?

...?

⌊ Introduction - 1 ⌋

- “new” – well, sort of.

  – The basic approach is ordered, inside-out, unconditional rewriting which certainly is not ”new”.

  – But there are various relevant modifications/extensions which are ”new”, or at least, their simultaneous combination is “new”.

- “simplifier” – again, well, sort of.

  – With this “new” procedure, the emphasis is placed on unconditional rewriting, but...

  – Several extensions to standard rewriting allow for new approaches to unconditional rewriting

⌊ **Background** ⌋

- Recently, Shankar (PVS) wrote a paper contrasting two approaches to automated deduction:

  – "Resolution"

    ∘ essentially described as procedures which "search" for a proof in an undecidable logic

  – "Decision Procedure"

    ∘ algorithms for determining a proof (or counterexample) within a decidable logic

- Shankar's argument was that "decision procedure" based approaches were far more viable in practice

  – User uses theorem-prover to reduce proof to subproblems in decidable logics.

⌊ **Background** ⌋

- Somewhere in-between "Resolution" and "Decision Procedures" lies user-extended rewriting

  – Rewriting-based simplifiers allow the user to extend simplification (in a sound manner) for finding proofs in undecidable logics.

  – This is basically the ACL2 approach.

  – Most other theorem provers also support user-extension through proven rewrite rules (even PVS)

- My previous efforts on several algorithms for simplifying terms for various goals led to a realization that each was an instance of a more general algorithm

⌊ ACL2 Sit. Rep. - 1 ⌋

- ACL2's built-in simplifier works well for a large class of theorems, but there are relevant issues with the ACL2 simplifier:

  - Efficiency...

  - Complexity...

  - Control...

- For many problems, these issues are irrelevant, but there are problems for which ACL2 is ill-suited.

⌊ ACL2 Sit. Rep. - 2 ⌋

- Possible remedy: include more hooks to efficient decision procedures for various sub-logics.

  – This would help efficiency, but at the definite cost of complexity and the almost certain cost of control.

  – Further: it is non-trivial to get decision procedures to integrate and cooperate efficiently and effectively.

- Proposed remedy: architect minimal, optimized, controllable, rewriter

  – Certainly can achieve control. Almost certainly less complex (and possibly provably sound). The real question is efficiency.

⌊ **Proposed *Architecture* - 1** ⌋

- A *kernel* architecture for a simplifier.
- The specification for the *kernel*:

  – responsible for maintianing the logical state

    ◦ soundness is completely determined by the *kernel*

  – code is optimized but as simple as possible

  – only **fundamental** necessary extensions are made to the kernel

  – provide as much control within the bounds of soundness to “user extensions”

⌊ **Proposed *Architecture* - 2** ⌋

- The user extends the system chiefly by proving rewrite rules.

  – Rewrite rules can have *sieves*

- The *sieve* processing function can:

  – determine whether a matching rewrite rule should be applied,

  – make *safe* changes the kernel logic state, and

  – be replaced by the user without affecting soundness

- The *kernel* in this case is an efficient ordered, inside-out, unconditional rewriter with "extensions"

⌊ **Example "extensions" - 1** ⌋

- Maintenance of Equal-ities through pointers

  – `n.rep = r` implies that in the current context, `(equal n r)` is true

  – *rep.* pointers are used for efficient equality reasoning and memoization of simplification results

  – *rep.* pointers are updated as a function of the *context*:

  ```
  (if x 't (if y 't (equal x y)))
  ```

- Masking of looping rewrite rules

  – Allows use of rules which would normally lead to infinite rewrite sequences:

  ```
  (equal (+ x y) (+ y x))
  ```

  – More effective and common use of rule masking is for rules where the `<lhs>` is a subterm of the `<rhs>`.

⌊ Example "extensions" - 2 ⌋

- Efficient allocation of terms.

  – No garbage collection, no `cons`-ing, no fixnum boxing, etc.

  – Terms begin as garbage are *promoted* on demand or by user-controlled heuristics.

- Unique construction of *promoted* terms.

  – syntactic equivalence is fast, constant-time operation

- Efficient management of contexts.

  – single logic-state stobj is updated through each context.

  – stratified `undo-stack` used to quickly revert only those *rep.* assignments relevant to the current context.

⌊ Why Ordered?, Inside-Out? - 1 ⌋

- Our experience with ACL2 should give us some intuition about the value of ordered, inside-out rewriting.

- A nice property about inside-out rewriting:

  – The union of two orthogonal terminating rewrite systems is inside-out terminating.

  – Termination is not "modular" in general (Toyama):

```
P := { (equal (f 0 1 x) (f x x x)) }
Q := { (equal (g x y) x), (equal (g x y) y) }
```

  – Infinite rewrite in `(union P Q)`:

```
(f (g 0 1) (g 0 1) (g 0 1)) -> ->
(f    0          1  (g 0 1)) ->
(f (g 0 1) (g 0 1) (g 0 1))
```

⌊ **Why Ordered?, Inside-Out? - 2** ⌋

• But what about the cases where we have wanted outside-in rewriting in the past, e.g. the NU-rewriter?

• The common application for the NU-rewriter is to efficiently apply the following rule:

```
(equal (nth i (update-nth j v x))
       (if (n= i j) v (nth i x)))
```

– The need for outside-in rewriting was due to the fact that `x` may be a non-trivial nest of `update-nth` terms

– I have found that inside-out rewriting is sufficient if disables each field update function and adds a rule of the form:

```
(equal (nth i (update-foo x s))
       (if (= i 2) (new-foo x s) (nth i x)))
```

⌊ **Why Unconditional? - 1** ⌋

• No conditional rewriting, What the Hell?!?, Have I Gone Completely Mad?

• First, unconditional rewriting affords more elegant, simpler, and efficient algorithms

• Second, with the *extensions*, one can "simulate" conditional rewriting. Assume we want to simulate the following rule:

```
(defthm foo-bar
  (implies (ok x)
           (equal (foo x) (bar x))))
```

⌊ Why Unconditional? - 2 ⌋

- ... `foo-bar` now becomes ...

```
(defun fb! (c x)
  (if c (bar x) (foo x)))
(in-theory (disable fb!))

(defthm foo-bar1
  (equal (foo x) (fb! (ok x) x)))

(defthm foo-bar2
  (equal (fb! c x)
         (if (equal c (ok x))
             (foo x)
           (fb! c x))))

(defthm foo-bar3
  (equal (fb! t x) (bar x)))
```

- Thank goodness for macros :)

⌊ But Wait, There's More... ⌋

- No free variables either!

- Another useful application of masking looping rules is to allow for rules which "pump" terms into larger terms.

- Common rule with free variables in the hypothesis:

```
(defthm <<-transitive
  (implies (band (<< x y) (<< y z))
           (<< x z)))
```

- Turn this into the following unconditional rule:

```
(defthm funky-<<-transitive
  (equal (band (<< x y) (<< y z))
         (band (<< x z)
               (band (<< x y) (<< y z)))))
```

⌊ Strengths and Weaknesses ⌋

- I have (or believe I will have) support for simulating the following:

```
BDDs,
equality reasoning,
AC matching,
conditional rewriting,        (well, sorta)
free-variable matching        (well, sorta)
NU-rewriter                   (well, sorta)
linear arithmetic             (well, sorta)
congruence rewriting          (well, sorta)
staged simplification
finite case-split rewriting
...
```

- Even cooler, if defined appropriately, these "simplifiers" can be easily and selectively composed

- What can't be simulated: optimized SAT checkers