Validated Construction of Congruence Closures

Aaron Stump

Computer Science and Engineering
Washington University
St. Louis, Missouri, USA

Disproving '05

Congruence Closure Algorithms

- Input: set E of equalities between ground terms
- Output: (finite representation of) least congruence relation extending E
- E.g.

  If $E = \{a = b, a = c, f(a) = d\}$,
  then congruence should include $\langle f(f(c)), f(d)\rangle$.
- Key ingredient in testing satisfiability of q.f. EUF formulas.
- EUF heavily used in verification.

Congruence Closure as Ground Completion

- Congruence closure can be viewed as ground completion [Bachmair Tiwari 2000, Kapur 1997]
- Input: set E of equalities between ground terms
- Output: convergent rewrite system R such that $=_R$ conservatively extends $=_E$
- E.g.

  $E = \{a = b, a = c, f(a) = d\}$

  $R = \{a \to b, b \to c, f(c) \to d\}$

  and then $f(f(c)) \downarrow_R f(d)$
- To test satisfiability of conjunction of literals
  - Build congruence closure of equations.
  - Test each disequation by putting lhs and rhs in normal form.

Validated Results from Congruence Closure

- CC algorithms (usually!) proved correct on paper.
- Bugs still possible in implementation.
- Raise confidence in results by emitting evidence:
  - For unsatisfiability, return a proof.
  - For satisfiability, return a model.
- Take the convergent rewrite systems as models.

Evidence-Producing ⇒ Partially Verified

Research Agenda

Move towards verified software by statically checking that evidence produced will always be well-formed.

- Type checking for evidence-manipulating code.
- Evidence from ok run will check.
- Advertisement: proposed IJCAR/FLoC workshop PLPV '06, "Programming Languages meets Program Verification"

Validated Congruence Closure Algorithms

- Previous work: validated proof production from CC [Klapper Stump 2004].
- Guarantee: ok run $\Rightarrow$ proof will check.
- Code written in our RSP1 dependently typed programming language [Westbrook Stump Wehrman 2005].
- The current work: validated model generation from CC.

Validated Model Generation from CC

- Emit convergent rewrite system R as model for equations E.
- Statically verify:
  - R will be convergent
  - $=_R$ will conservatively extend $=_E$
  - R will satisfy the disequations (if reported so)
- The implementation manipulates proofs of these properties.
- Actually: proofs of sufficient conditions.
- Emitted proofs can be independently checked.
- Shostak's algorithm, in Abstract Congruence Closure framework.

Rest of the Talk: Implementation in RSP1

1. Quick intro to RSP1
2. The CC data structure
3. Simplification phase (rewriting)
4. Extension phase (introduction of new constants)
5. Other phases still future work!

Dependently Typed Programming in RSP1

- Small functional language with dependent types.
- Like in ML: user-declared datatypes, pattern matching, recursion.
- Unlike: datatypes can be term-indexed.
- Programmer can declare a datatype of proofs indexed by the formula proved.
- Instead of `pf` we have `pf f`, where `f` is the formula proved.
- Type checking ensures proofs well-formed.

Terms and Equations

```
i :: type;;
injconst :: c:const => i;;
apply :: n : nat => func n => ilist n => i;;

o :: type;;
equals :: i => i => o;;
```

`const`: type for new consts, involves a trick.
`func n`: type for functions of arity n.
`ilist n`: type for lists of terms of length.

Datatype for CCs

Three components for a CC problem (in abstract CC framework):

- Equations still to process
- C-rules $c \rightarrow d$, where c, d new constants
- D-rules $f(\bar{c}) \rightarrow d$, where $\bar{c}, d$ new consts
- We will maintain the invariant: no overlaps at all.

```
cc_t :: type;;
mkcc :: olist => l:crlist => drlist l => cc_t;;
```

Lists of C-Rules

```
crlist :: type;;
crn :: crlist;;
crc :: c2:const =>
       c1:const =>
       gtc c2 c1 =>
       l:crlist =>
       const_apart c2 l =>
       const_apart c1 l =>
       crlist;;
```

`gtc c2 c1`: `c2` is bigger than `c1`.
`const_apart c2 l`: `c2` is not on the lhs of any rule in `l`.

Lists of D-Rules

$f(\bar{c}) \rightarrow d$

```
drlist :: crlist => type;;
drn :: l:crlist => drlist l;;
drc :: n:nat =>
       f:func n =>
       cs:clist n =>
       d:const =>
       l:crlist =>
       L:drlist l =>
       A:const_apart d l =>
       T:term_apart n f cs l L =>
       As:const_list_apart n cs l =>
       drlist l;;
```

`clist n`: type for lists of consts of length n.
`term_apart n f cs l L`: type for proofs that `f(cs)` is not on the lhs of `L`'s rules.

Note: the Intrinsic Style

- To build lists of C-rules and lists of D-rules, proofs required.
- Cannot build a CC which is not convergent.
- Extrinsic style would keep proofs separate from data.

Simplification and Extension

- Simplification rewrites terms using C- and D-rules.
- Extension introduces new consts for non-const subterms.
- How to guarantee freshness of consts?
  - Associate numbers with consts.
  - Keep numeric bounds on all rules.
  - Code must manipulate proofs of boundedness.

Simplification

```
rec
simplify :: l:crlist =c>
            e:olist =c>
            L:drlist l =c>
            b1:nat =c>
            bound_crlist b1 l =c>
            bound_drlist b1 l L =c>
            q:{x:i, B:bound_term b1 x} =c>
            {y:i,
             D:provese (mkcc e l L) (equals q.x y),
             C:canonical y l L,
             B:bound_term b1 y} = ...
```

`provese cc f`: cc proves formula f.

`canonical y l L`: y is in canoncial form.

Extension

```
rec
extend :: l:crlist =c>
          e:olist =c>
          L:drlist l =c>
          b1:nat =c>
          bound_crlist b1 l =c>
          bound_drlist b1 l L =c>
          q:{x : i, C:canonical x l L, D: bound_term b1 x} =c>
          {c:const,
           z:nat,
           aa:assoc_num z c,
           b:nat,
           g1:gte b b1,
           g2:gt b z,
           L2:drlist l,
           B1:bound_crlist b l,
           B2:bound_drlist b l L2,
           d1:provescc (mkcc e l L) (mkcc e l L2),
           d2:provescc (mkcc e l L2) (mkcc e l L),
           d3:provese (mkcc e l L2) (equals q.x (injconst c)),
           A1:const_apart c l,
           A2:const_apart2 c l L2} = ...
```

Conservativity

Trick: make constants logically transparent:

```
const :: type;;
mkcanon :: i => nat => const;;
peSpecial :: cc:cc_t => t:i => n:nat =>
             provese cc
               (equals t (injconst (mkcanon t n)));;
```

Conclusion

- Work in progress validating model generation from CC.
- Models are convergent rewrite systems.
- Code builds proofs showing convergence, conservativity.
- Type checking in RSP1 guarantees those properties.
- Main future work: finish implementation.