Validated Construction of Congruence Closures

Aaron Stump
Dept. of Computer Science and Engineering
Washington University in St. Louis
St. Louis, Missouri, USA
Web: `http://www.cse.wustl.edu/~stump/`

June 23, 2005

Abstract

It is by now well known that congruence closure (CC) algorithms can be viewed as implementing ground completion: given a set of ground equations, the CC algorithm computes a convergent rewrite system whose equational theory conservatively extends that of the original set of equations. We call such a rewrite system a CC for the original set. This paper describes work in progress to create an implementation of a CC algorithm which is validated, in the following sense. Any non-aborting, terminating run of the implementation is guaranteed to produce a CC for the input set of equations. Note that aborting or failing to terminate can happen for implementations of CC algorithms only due to bugs in code; the algorithms themselves are usually proved terminating and correct. Validation of an implementation of a CC algorithm is achieved by implementing the algorithm in RSP1, a dependently typed programming language. Type checking ensures that proofs of convergence and conservative extension are well-formed.

1 Introduction

Thanks to work of Kapur and Bachmair and Tiwari, it is now clear that congruence closure (CC) algorithms can be viewed as a form of ground completion [5, 2]. The cited works show that many congruence closure algorithms from the literature can be viewed as constructing a convergent rewrite system for an input set of ground equations. The rewrite system is expressed over an extension of the signature for the input equations. Hence, the equational theory of the rewrite system produced by the CC algorithm is a conservative extension of the equational theory of the input equations, but generally not equivalent.

There is ongoing interest in the automated reasoning community in validity checking tools that can produce independently checkable evidence for the results they report. When a tool reports a formula to be valid, the evidence is a proof of validity. When a tool reports a formula to be invalid, the evidence is a countermodel. Producing proofs is important for exporting results from validity checkers to proof assistants, and for some approaches to applications like proof-carrying code [3, 4, 7]. Producing independently checkable proofs can also increase confidence in the results of the validity checker, which is often a highly optimized and complex piece of software. Proof production can also be used to increase performance of certain kinds of validity checkers (see [8] and works cited there).

In previous work, Rob Klapper and I describe how to implement certain proof-producing decision procedures, including one based on CC, which are statically validated in the following sense [6]. Any non-aborting, terminating run of the decision procedure that reports the input formula valid is guaranteed to produce a well-formed proof of validity for that formula. Note that the issue here is the possibility of bugs in the implementation of the proof-producing decision procedure. The decision procedure itself, as an algorithm, has been proven (on paper) to terminate and produce a well-formed proof if the input formula is valid. But it is all too easy in mainstream programming languages to write code which accidentally produces ill-formed proofs. Tracking down the sources of ill-formed proofs can be extremely time-consuming, particularly for large input formulas. With validated proof production, such bugs never arise: the proofs are guaranteed to check.

Our approach to achieving a validated implementation is to implement the decision procedure in RSP1, an imperative programming language with dependent types [9]. Leaving aside the imperative features, which are not used in this paper, RSP1 can be thought of as a dependently typed version of the core functional part of a language like Ocaml. Just as in Ocaml, datatypes can be declared by the user. Unlike in Ocaml, these datatypes can be indexed by terms. So instead of having just a datatype of proofs, we can have a datatype of proofs indexed by the formula (encoded as an element of a datatype of formulas) which is proved. Proofs of encoded formula `phi` have type `pf phi`. A function's expectation of a proof of a particular formula can thus be recorded in a type, and compile-time type checking then ensures that proofs are manipulated in a type-safe way. Pattern-matching constructs are available, just like in Ocaml, but are dependently typed to enable manipulation of term-indexed datatypes. A compiler for RSP1 to Ocaml has been implemented, enabling reasonably fast execution of code validated by the RSP1 type checker. RSP1 currently lacks parametric polymorphism, so some datatypes, notably lists, must have different versions for different types of constituent data.

Our previous work concerns validated proof production from congruence closure (and other automated reasoning algorithms). The current paper is concerned with validated model generation from a CC algorithm. In particular, work in progress is described to implement a validated version of Shostak's algorithm, as cast in the framework of Abstract Congruence Closure [2]. The implementation explicitly manipulates proofs showing the rewrite system constructed by the algorithm is convergent and has an equational theory conservatively extending the ground equations supplied as input. It is not our goal to develop a formal theory of convergence from first principles (see instead, e.g., [1]). Instead, we take the classic results of such a theory for granted, and simply seek to establish statically that conditions sufficient (by that theory) for convergence hold. The proofs of these conditions can be produced by the implementation and independently checked. But as for the previous implementation of validated proof-producing congruence closure, the implementation is done in RSP1, and RSP1 type checking statically ensures that those conditions will hold for the CCs produced. It has turned out that getting an implementation with validated model generation has required a different, more intricate approach to the CC algorithm than was necessary for validated proof production. In particular, it has proven useful to model the data structures used in Abstract Congruence Closure much more faithfully than was necessary to get validated proof production. Hence, the implementation described below is done from scratch, without any code or proof reuse from the earlier implementation.

What use is it to have model generation from a CC algorithm statically validated? After all, it is relatively inexpensive to check that the (ground) rewrite system produced (the CC) is convergent, particularly since the algorithm produces a shallow system: all right hand sides of rules are constant symbols and all left hand sides are either constants or applications of a function symbol to a list of constant symbols as arguments. It can also be easily checked that the CC produced entails the original equations. It is not immediately obvious how to check that the equational theory of the CC is a conservative extension of the equational theory of the original equations, although perhaps this can be done. So having statically validated model generation may not greatly increase confidence in the individual results reported by the implementation. It certainly should increase confidence in the correctness of the implementation itself. Furthermore, having an implementation like the one (in progress) described in this paper actually implemented in a proof assistant based on dependent type theory, like Coq, would confer an additional benefit: the proof assistant could trust that any CC produced by the implementation was correct, without actually having to build any of the proofs. Type checking shows that the proofs would check if produced. There are a few places in the current implementation where some modest changes would be required to support this. In particular, there are a few places where a type checker cannot easily see that the code cannot fail. It should be possible to eliminate these, but it proved more convenient not to insist on avoiding all such situations here.

In the rest of the paper, the current implementation in progress is described. This implementation comprises 2000 lines of RSP1, including a number of currently unproved lemmas. In the setting of this paper, the implementation is presented at the level of datatypes and function specifications. Hence, familiarity with the syntax of RSP1 is not necessary for reading the rest of the paper. Detailed knowledge of BT (I will use this abbreviation from now on to refer to [2]) is also not required, though it will be useful. The implementation currently comprises the simplification and extension phases of Shostak's CC algorithm (in the terminology of Abstract Congruence Closure). The latter is nontrivial in this context, since it is where new constant symbols are introduced, and hence where conservative extension must be shown. The work yet remaining to be done is admittedly substantial: orientation, deletion, deduction, collapse, and composition must be implemented. Nevertheless, many important issues show up just in simplification and extension, including design of the datatype for the CC, with which we begin.

2 The Datatype for CCs

Abstract Congruence Closure (ACC) problems consist of a set of equations to be processed and the convergent rewrite system resulting from the processing so far. In addition, in BT, the set of new constant symbols introduced so far is also part of an ACC problem, but we maintain information about the new constants in a different way, discussed below. As mentioned above, the rewrite system is shallow. There are two kinds of rewrite rules. *C-rules* are of the form $c \rightarrow d$, where c and d are constant symbols not occuring in the original input equations. *D-rules* are of the form $f(c_1, \ldots, c_n) \rightarrow d$, where $c_1, \ldots, c_n$ and d are all new constant symbols not occurring in the original equations. Note that n may be 0 in this case, to map a constant from the original equations to a new constant. Hence, in the implementation, we take the following definitions for ACC problems:

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

These declare that `cc_t` is a type, and that to form one, using the term constructor `mkcc`, you must supply three things. The first is an `olist`, whose declaration as the datatype of lists of formulas is omitted here. Also omitted is the simple declaration of the datatype `o` of formulas. The second item needed by `mkcc` is an element of the type `crlist`, which we declare (see below) as the datatype for lists of C-rules. The third item is a `drlist l`, which is a list of D-rules. The index `l` in the type `drlist l` indicates, as we shall see below, that no constants used in any D-rule in the list appears as the left hand side of a C-rule in `l`.

2.1 The Datatype for Lists of C-rules

Lists of C-rules may be built using the datatype determined by the following declarations:

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

The empty list of C-rules is formed using the 0-ary constructor `crn`. To add a C-rule to an existing `crlist`, the constructor `crc` is used. It requires the left and right hand sides (`c2` and `c1`, respectively) of the C-rule. We declare `const` as the type for new constant symbols; the definition involves a trick, and is postponed to the discussion of conservative extension below. The constructor `crc` next requires a proof that `c2` is greater than `c1` in a certain basic ordering on the new constant symbols. This requirement is taken from BT. We will associate natural numbers with `consts`, and then order `consts` by number (discussed below). Next, `crc` requires the `crlist` to which the C-rule `c2` $\rightarrow$ `c1` is to be added. Finally, proofs that `c2` and `c1` are *apart* from `l` are required. The intended meaning of `const_apart c l` for any `c` and `l` is that `c` does not appear as the left hand side of any rule in the list of C-rules `l`. The rules for `const_apart` are straightforward, although they rely on an auxiliary judgment `neqc` that two `consts` are distinct.

So when a list of C-rules is built, it is guaranteed to be convergent: the left hand side of each rule is less than the right hand side, and no `const` appears on the left hand side of two C-rules in the list. We do not formally express in our RSP1 implementation the property of being convergent. As remarked above, developing a full formal theory of convergent rewrite systems is beyond the scope of this project. Hence, we formally express other conditions, which are sufficient for convergence. The proof of sufficiency is done outside RSP1, on paper.

2.2 The Datatype for Lists of D-rules

As remarked at the start of this Section, the type for lists of D-rules all of whose `consts` are apart from a list `l` of C-rules (where a `const` is apart from a C-rule if it is different from that C-rule's left hand side) is `drlist l`:

```
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;;
```

The first declaration says that `drlist` is a datatype indexed by `crlists`. For any `crlist l`, the empty list of D-rules apart from `l` can be formed using the constructor `drn`. To add a D-rule to an existing list of D-rules, the constructor `drc` is used. Recall that a D-rule is of the form $f(c_1, \ldots, c_n) \to d$, where $c_1, \ldots, c_n, d$ are new constant symbols (`consts`) not occurring in the original input equations. The first four arguments to `drc` are all the constituent pieces of the D-rule. The type `func n` is for function symbol of (fixed single) arity `n`, which is declared to be of type `nat`. The latter is the standard datatype for natural numbers in unary, with constructors `z` (for zero) and `s` (for successor). The type `clist n` is the type for lists of length `n` of `consts`. Then `drc` requires an `l` which is a `crlist`, and the existing `drlist l` to which to add the new D-rule. BT shows termination is preserved in this situation, since each such new D-rule is contained in a natural reduction ordering. To ensure local confluence, `drc` requires several proofs about items' being `apart`. All the `consts` in the new D-rule must be apart from `l`, which is expressed in the types for arguments `A` and `As`. And the left hand side, $f(c_1, \ldots, c_n)$, of the new D-rule is required to be different from the left hand side of any D-rule in the existing list of D-rules (`L`). The declarations for `term_apart` and `const_list_apart` are unsurprising and omitted here.

2.3 Datatypes for Terms and Lists of Terms

Our implementation of Shostak's CC algorithm processes equations between terms. Terms are declared as follows:

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

A term, of type `i`, is either an application of a function symbol of arity `n` (`func n`) to a list of `n` terms (`ilist n`); or an injection of one of our new constant symbols. The datatype for lists of terms is declared as follows. Note that the type `ilist` for such lists is indexed by a `nat`, which gives the length of the list (this is a standard trick in dependently typed programming):

```
ilist :: nat => type;;
ilistn :: ilist z;;
ilistc :: i => n : nat => ilist n => ilist (s n);;
```

2.4 The Intrinsic Style

The style of encoding used here for CCs is what we might call the intrinsic style. Datatypes whose elements are intended to have some property are declared in such a way that only elements which have the property can actually be constructed. This is because the constructors take in proofs of all the required properties. Here, the properties are those which show convergence (apartness of constant symbols and left hand sides of D-rules, and containment of C-rules in the basic well-founded ordering on constants). We cannot form an element of type `cc_t` which does not have those properties. Hence, a certain kind of soundness is built right in to the datatype for CCs. This is a strong protection against soundness bugs. Unfortunately, it also seems to complicate the rest of the implementation substantially, since every time a CC or constituent part of one must be manipulated, many proofs are required. Some of these proofs might not be essential to the soundness of the operation in question, but they must typically be supplied anyway. In contrast, an extrinsic style would not require proofs to construct elements of a datatype like `cc_t`. The proofs would be kept completely separate from the data, and passed around as additional arguments as necessary. It would be interesting to try the implementation again in the extrinsic style, but for the time being, we forge ahead intrinsically to the simplification and extension phases of Shostak's CC algorithm in RSP1.

3 Simplification and Extension

The simplification phase of Shostak's CC algorithm, in the Abstract Congruence Closure framework, is intended to put terms into canonical form with respect to the current list of C-rules and D-rules computed thus far. Our implementation just uses linear search through the lists of C-rules and D-rules to find a match, simplifying terms bottom-up; it is conceivable that an indexing data structure could be used for better efficiency.

```
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} = ...
```

Figure 1: Declaration for `simplify`

After a term has been put into canonical form by simplification, it is handed off to the extension phase. Extension introduces new constant symbols bottom-up for every subterm of the input term which is not already the injection of a `const`. At the end of extension, the input term has been reduced to a single `const`, and new D-rules of the form $t \rightarrow d$ have been added for any subterm t of the input term for which a new `const` d was introduced. Since such `const`s must be fresh, some mechanism is needed to enable simplification to keep track of what the next `const` to be introduced may safely be. The mechanism used here is to associate a number with each `const`, and then bound the set of `const`s used in the C-rules and D-rules. The next fresh `const` to generate may safely be any that has an associated number greater than the bound on the `const`s already used by the C-rules and D-rules. Both simplification and extension require fairly elaborate helper functions to process lists of arguments in applications. Space limitations prevent further discussion of these, which are essentially the natural extensions of simplification and extension to lists of terms.

3.1 Simplification

The declaration for `simplify`, which implements simplification, is given in Figure 1. This declaration, whose body has been omitted ("`...`"), says that `simplify` is a recursive computational function. The symbol `=c>` (as opposed to `=>`) is used in RSP1 to indicate that a function is computational and may pattern match on its argument. The other function space (`=>`) is used for the types of term constructors. The notation `q:{x:i, B:bound_term b1 x}` declares that argument `q` is a dependent record consisting of a term `x` and a proof term `B` of type `bound_term b1 x` (more on this shortly). The function `simplify` takes in all its arguments, and returns a record of resulting values ("`{y:i, ... }`").

Let us look at the arguments and the resulting values to `simplify`. The first three arguments, `l`, `e`, and `L` are the constituent pieces of the CC with respect to which `simplify` is supposed to rewrite a term. That term is given by the `x` field of the argument `q`, which, as just explained, is a dependent record. In addition to the CC and the term to rewrite using that CC, `simplify` requires proofs that all the `consts` occurring in several different entities are bounded. That is, the numbers associated with those `consts` are less than the bound, which is the `nat` number `b1`. We require for simplification a proof that the `consts` in the C-rules and the D-rules are bounded (`bound_crlist b1 l` and `bound_drlist b1 l L`, respectively). We also require a proof that all the `consts` appearing in the term `x` are bounded by `b1` (`bound_term b1 x`). This will enable us to prove that any term returned by simplification has all its `consts` bounded by that same bound; this is expressed by the field `B` in the record returned by `simplify`. That record also contains a field `y` for the canonical form that `simplify` computes for `x`, and a proof (field `D`) that the CC implies that `x` equals `y` (proof rules for the `provese` judgment, that a CC derives a single formula, are omitted here for space reasons). Finally, the record returned by `simplify` has a field `C` for a proof that the canonical form `y` is indeed canonical with respect to the C-rules and D-rules supplied.

3.2 Extension

Figure 2 gives the declaration for `extend`, which implements extension. This function takes in the same first three arguments as `simplify`, which are the constituent parts of the CC as it stands before extension. Since `extend` may add new D-rules to the CC, it returns a new `drlist`, as field `L2` of the returned record. As `simplify` did, `extend` also takes in a bound `b1` on the `consts` occurring in the lists of C-rules and D-rules. The record `q` required as the last argument contains the term `x` to extend, a proof `C` that `x` is canonical, and a proof that all the `consts` occurring in `x` are bound by `b1`. As discussed above, the latter two proofs are produced by `simplify` so they may be provided to `extend`.

The record of values returned by `extend` returns quite an assortment of different proofs for the different invariants maintained by the code. First, extension always produces a `const` as its result, which is returned in the field `c`. This `const`, like all `consts`, has an associated number, which is returned as the field `z`. A proof certifying the association is also returned (field `aa`). Since extension may introduce new constants, a new bound must be produced on the `consts` occuring in the list of C-rules and the updated list (`L2`) of D-rules. This bound is returned in the field `b`, and proofs of the new bounds on the lists of C-rules and D-rules are returned in fields `B1` and `B2`.

Finally, we come to the proofs (`d1` and `d2`) that the old CC is equivalent to the new CC. Proof rules for the judgment `provescc` are omitted here, but they say, naturally enough, that one CC entails another if it entails all the other's equations, C-rules, and D-rules. So we are insisting here that the equational theories of the two CCs are equivalent, which seems incompatible with the fact that the new CC may be just a conservative extension of the starting CC, due to the introduction of new constants. We return to this point shortly, but first comment on the last of the returned values. The proof returned in field `d3` shows that the new CC entails that the input term (`q.x`) equals the (injection of the) returned constant `c`. The proofs `A1` and `A2` show that the returned `const` is apart from the C-rules and is not used in the left hand side of any D-rule in the new CC, respectively. The latter is needed to show that the D-rule obtained by non-trivially extending the arguments of an application does not have a left hand side already occurring in the list of D-rules.

```
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} = ...
```

Figure 2: Declaration for `extend`

Finally we come to the issue of conservative extension. The argument for conservative extension in BT proceeds by induction on the form of proofs of equalities between terms without newly introduced constants that can be conducted in the new CC. It shows how to transform such proofs into ones which can be conducted in the old CC. Returning such a proof from `extend` would (most naturally) require returning a RSP1 function representing the inductive argument. While possible in RSP1, this is a bit outside the current programming methodology. We would prefer to return just an element of a datatype for a proof, instead of an RSP1 function.

The trick we use for this comes in the declarations for `const`, and the associated proof rule:

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

We introduce new `const`s with the constructor `mkcanon`. The trick is that we index new constants with the term they are intended to represent in the extension of the CC. So `mkcanon t n` is the `const`, with associated number `n`, representing term `t` in the extension. The `peSpecial` proof rule then just says that logically, the `mkcanon` constructor is transparent: `mkcanon` expressions equal the terms (`t`) they are intended to represent. Hence, the equational theories are the same, even though we introduce new constants.

References

[1] CoLoR: a Coq Library on Rewriting and Termination. Available at http://color.loria.fr, 2005.

[2] L. Bachmair and A. Tiwari. Abstract Congruence Closure and Specializations. In David McAllester, editor, *17th International Conference on Automated Deduction*, volume 1831 of *LNAI*, pages 64–78. Springer-Verlag, 2000.

[3] E. Contejean and P. Corbineau. Reflecting Proofs in First-Order Logic with Equality. In R. Nieuwenhuis, editor, *20th International Conference on Automated Deduction*, 2005.

[4] E. Deplagne, C. Kirchner, H. Kirchner, and Q. Nguyen. Proof Search and Proof Check for Equational and Inductive Theorems. In F. Baader, editor, *Conference on Automated Deduction - CADE-19, Miami, USA*, 2003.

[5] D. Kapur. Shostak's congruence closure as completion. In H. Comon, editor, *8th International Conference on Rewriting Techniques and Applications*, pages 23–37. Springer-Verlag, 1997.

[6] R. Klapper and A. Stump. Validated Proof-Producing Decision Procedures. In C. Tinelli and S. Ranise, editors, *2nd International Workshop on Pragmatics of Decision Procedures in Automated Reasoning*, 2004.

[7] G. Necula. Proof-Carrying Code. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, January 1997.

[8] A. Stump and L.-Y. Tan. The Algebra of Equality Proofs. In Jürgen Giesl, editor, *16th International Conference on Rewriting Techniques and Applications*, 2005.

[9] E. Westbrook and A. Stump. A Language-based Approach to Functionally Correct Imperative Programming. 10th ACM SIGPLAN International Conference on Functional Programming, 2005.