CADE-20

Workshop on Disproving - Non-Theorems, Non-Validity, Non-Provability

Tallinn, Estonia, July 22, 2005

Wolfgang Ahrendt, Peter Baumgartner, Hans de Nivelle
(Organizers)

Preface

The *Disproving Workshop* was held at the 20th International Conference on Automated Deduction, in Tallinn, Estonia. The name *automated theorem proving* or *automated deduction* derives from the fact that the field traditionally focussed on the art of automatically finding proofs. Initially, researchers were mainly motivated by the wish to build computer systems that could automatically solve hard, mathematical problems. When searching for a very hard proof, it is quite acceptable for a system to eat up all resources and to never to give up. After all that is what we, researchers are also doing all the time.

However in the last years, one has become aware of the fact that for many applications, one needs to take more of an engineer's approach. In particular, one needs to be aware of resources. In order to use resources efficiently, it is essential to be able to efficiently recognize non-theorems. As an example, consider a situation where an automated theorem proving system is used as assistant for automatically solving subtasks in a larger, interactive project. In this context, the requirements to the automated theorem prover are quite different than in mathematics. In case, the user is working on a faulty conjecture, he should find out as early as possible. In addition, in case it cannot find a proof, the prover should provide as much information as possible, so that the user can correct the conjecture. The papers collected in this volume address this topic from different angles.

M. Demba, F. Alexandre and K. Bsaïes study how to correct faulty (universally quantified) conjectures by adding additional assumptions. The method is based on folding/unfolding rules.

In the paper by Ph. Rümmer, the problem of obtaining counter examples from invalid formulas in Java Dynamic Logic is studied. Dynamic Logic is an extension of first-order logic which can reason about programs in a natural way. A subfragment of Java Dynamic Logic is defined, for which the invalidity problem can be reduced to the validity problem. Using this, a verification condition can be proven incorrect.

M. Bezem describes experiments of a more mathematical nature: A geometric logic prover is adopted to find the minimal counter models against the assumption that all lattices are distributive.

A. Stump presents an implementation in progress of the congruence closure algorithm. The implementation takes place in a new programming language RSP1, which has a typesystem so rich that it can express correctness conditions as types. As a consequence, correctness of the program follows from its type correctness.

In addition to the contributed papers, there are two abstracts by the invited speakers. The first invited speaker, J. Giesl, discusses methods for proving non-termination of term rewrite systems using the dependency pair method. The method can prove non-termination of the term rewrite system under the innermost reduction strategy. This is particularly important, because innermost reduction corresponds to the execution model of standard programming langauges. The second invited speaker, B. Cook, describes

the use of automated reasoning tools at Microsoft.

We are indebted to the members of the program committee for their reviewing efforts. We made use of the *Easy Chair* system by Andrei Voronkov, which simplified the effort of organizing the reviewing process. Special thanks also deserve our invited speakers Jürgen Giesl and Byron Cook (the latter being joint invited speaker with the ESCAR workshop).

Wolfgang Ahrendt,
Peter Baumgartner,
Hans de Nivelle

Members of the program committee:

- Wolfgang Ahrendt, Department of Computing Science, Chalmers University of Technology, Göteborg, Sweden.
- Peter Baumgartner, Max-Planck Institut für Informatik, Saarbrücken, Germany.
- Johan Bos, University of Edinburgh, Scotland, United Kingdom.
- Chris Fermüller, Technische Universität Wien, Vienna, Austria.
- Ulrich Furbach, AI Research Group, University of Koblenz-Landau, Germany.
- Bernhard Gramlich, Technische Universität Wien, Vienna, Austria.
- Bill McCune, Mathematics and Computer Science Division, Argonne National Laboratory, Chicago, USA.
- Hans de Nivelle, Max-Planck Institut für Informatik, Saarbrücken, Germany.
- Harald Ruess, Computer Science Laboratory, SRI International, Menlo Park, USA.
- Renate Schmidt, School of Computer Science, University of Manchester, Manchester, UK.
- Carsten Schürmann, Department of Computer Science, Yale University, New Haven, USA.
- Graham Steel, Mathematics Reasoning Group, Centre for Intelligent Systems and their Applications, School of Informatics, Edinburgh, UK.
- Cesare Tinelli, Department of Computer Science, University of Iowa, Iowa City, USA.
- Andrei Voronkov, Department of Computer Science, University of Manchester, Manchester, UK.
- Calogero Zarba, Department of Computer Science University of New Mexico, USA.

Contents

Disproving Termination of Term Rewriting*

Jürgen Giesl, René Thiemann, Peter Schneider-Kamp

LuFG Informatik II, RWTH Aachen, Ahornstr. 55, 52074 Aachen, Germany
{giesl|thiemann|psk}@informatik.rwth-aachen.de

We focus on the termination analysis of *term rewrite systems* (TRSs) [1], since term rewriting provides a general mechanism to model evaluation in different programming languages. Therefore, techniques for termination analysis of TRSs can often be adapted to other programming languages afterwards. However, almost all existing techniques for automated termination analysis try to prove *termination* and there are hardly any methods to prove *non-termination*.

In this work, we introduce techniques to *disprove* termination of TRSs within the so-called *dependency pair framework* [3, 4]. Apart from disproving *full* termination, we also present new methods which can disprove termination under the *innermost* evaluation strategy (i.e., they can disprove *innermost termination*). Innermost termination is particularly important in practice, since it corresponds to termination under the eager call-by-value evaluation strategy used in many programming languages.

The benefits of our results are twofold. First, detecting non-termination automatically can be very helpful for software development when debugging programs. Second, we show that combining termination and non-termination techniques within the dependency pair framework is particularly useful: On the one hand, termination techniques also help for disproving termination, because they identify those parts of a TRS which may cause non-termination. On the other hand, non-termination techniques are helpful for proving termination, because they can detect "dead ends" during a termination proof attempt.

We implemented and evaluated our contributions in the automated termination prover AProVE [2]. Due to these results, AProVE was the winning tool in the *International Competition of Termination Provers* 2005 [5], both for proving and for disproving (innermost) termination of term rewriting.

References

1. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge, 1998.
2. J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Automated termination proofs with AProVE. In *Proc. RTA '04*, LNCS 3091, pages 210–220, 2004.
3. J. Giesl, R. Thiemann, and P. Schneider-Kamp. The dependency pair framework: Combining techniques for automated termination proofs. In *Proc. LPAR '04*, LNAI 3452, pages 301–331, 2005.
4. J. Giesl, R. Thiemann, and P. Schneider-Kamp. Proving and disproving termination of higher-order functions. In *Proc. FroCoS '05*, LNAI, 2005. To appear.
5. International Competition of Termination Provers 2005. http://www.lri.fr/~marche/termination-competition/

* Supported by the Deutsche Forschungsgemeinschaft DFG under grant GI 274/5-1.

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/

July 1, 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, dis-

cussed below. As mentioned above, the rewrite system is shallow. There are two kinds of rewrite rules. *C-rules* are of the form $c \to d$, where c and d are constant symbols not occuring in the original input equations. *D-rules* are of the form $f(c_1, \ldots, c_n) \to d$, 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 `const`s, and then order `const`s by number (discussed below). Next, `crc` requires the `crlist` to which the C-rule `c2` $\to$ `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 $t \to d$ 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 `const`s occurring in several different entities are bounded. That is, the numbers associated with those `const`s are less than the bound, which is the `nat` number `b1`. We require for simplification a proof that the `const`s 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 `const`s 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 `const`s 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 `const`s 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 `const`s 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 `const`s, 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 `const`s 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.

Correction of faulty conjectures and programs extraction

M. Demba[1], F. Alexandre[2], and K. Bsaïes[1]

[1] Faculté des Sciences de Tunis, DSI
Campus Universitaire 2092 Tunis, Tunisie
{moussa.demba,khaled.bsaies}@fst.rnu.tn
[2] LORIA BP 239
54506 Vandoeuvre-lès-Nancy, France
alexandr@loria.fr

Abstract. We present a method for patching faulty conjectures in automatic theorem proving. The method is based on well-known folding/unfolding rules [2]. The conjectures we are interested in here are implicative formulas that are of the following form: $\forall \overline{x}\ (\exists \overline{Y}\ \Gamma(\overline{x}, \overline{Y}) \leftarrow \Delta(\overline{x}))$ where Γ and Δ are conjunction of atoms and no variable of $\overline{Y}$ occurs in Δ. A faulty conjecture is a statement $\forall \overline{x}\ \varphi(\overline{x})$, which is not provable in some given definite logic program $\mathcal{P}$, i.e, $\mathcal{M}(\mathcal{P}) \not\models \varphi$, where $\mathcal{M}(\mathcal{P})$ means the least Herbrand model of $\mathcal{P}$, but it would be if enough conditions, say P, were assumed to hold, i.e., $\mathcal{M}(\mathcal{P} \cup \mathcal{Q}) \models (\varphi \leftarrow P)$, where $\mathcal{Q}$ is a definition of P.
Key words: Predicate synthesis, program synthesis, inductive theorem proving, implicative formulas, folding/unfolding, abduction.

1 Introduction

We consider a theory $\mathcal{T}$ of definite logic programs (definite clauses) and implicative formulas. Given a conjecture φ :$\forall \overline{x}\ (\exists \overline{Y}\ \Gamma(\overline{x}, \overline{Y}) \leftarrow \Delta(\overline{x}))$ such that $\mathcal{M}(\mathcal{T}) \not\models \forall \overline{x}\ (\exists \overline{Y}\ \Gamma(\overline{x}, \overline{Y}) \leftarrow \Delta(\overline{x}))$, our aim is to turn φ into a theorem, by inserting assumptions into the right-hand-side of φ. An assumption is represented by a predicate say P defined by some program $\mathcal{P}$. P is said to be a corrective predicate of φ if we have $\mathcal{M}(\mathcal{T} \cup \mathcal{P}) \models \forall \overline{x} \forall \overline{y}\ (\Gamma(\overline{x}, \overline{y}) \leftarrow \Delta(\overline{x}), P(\overline{x}, \overline{y}))$. To synthesize $\mathcal{P}$ we exploit the information derived from a failed proof attempt of φ using the proof-as-program paradigm. This kind of predicate synthesis formalizes the problem of abduction [11, 8].

Let us consider the specification for the subtraction function in natural numbers: given two natural numbers v and w, find U such that $v + U = w$. To this specification corresponds the implicative formula:

$$\forall v \forall w(\exists Uplus(v, U, w) \leftarrow nat(v), nat(w))$$

which is false, as we discover while attempting to prove it, for example there is no U verifying $2 + U = 1$. Nevertheless, there are particular values for the universally quantified variables for which the formula is true. In general, a theorem prover will do nothing more but reject this conjecture. However, in many cases it is strongly interesting to know why the conjecture is false, and how it can be corrected. One can expect that the formula is valid on the assumption that $v \leq w$. The incomplete proof of this conjecture can be mapped into a corrective predicate P which is here the relation $\leq$ over natural numbers. In this paper we present a method which provides a computable definition of P.

This paper extends the use of corrective predicates considered in [3] by adding the rule of structural induction to deal with generalization technique.

Throughout the paper, Γ, Δ and Λ denote conjunctions of atoms; φ and π denote implicative formulas; A and B denote atoms, and θ and σ denote substitutions. Afterwards, existentially quantified variables are distinguished from universal variables by giving them upper-case letters, and the variables of the form $?x$ are called meta-variables. mgu means most general unifier and $< \pi_i \mid P_i >$ denotes the formula π_i and its corrective predicate P_i.

The rest of the paper is organized as follows: section 2 presents the inference rules, in section 3 we present the patching process and in section 4 we present some related works.

2 Inference rules

We prove conjectures containing existential quantifiers while providing explanations of the failures. Each inference rule is associated with a procedure construction of corrective predicates. The synthesized program is then the set of definite clauses generated during the proof attempt.

Definition 1 (Partial correctness). *Let $\varphi \ : \ \Gamma(\overline{x}, \overline{Y}) \leftarrow \Delta(\overline{x})$ be an implicative formula whose predicates are defined by the program $\mathcal{P}$. Let $\mathcal{Q}$ be a program defining a predicate P. The program $\mathcal{Q}$ is partially correct for $\mathcal{P}$ with respect to φ iff $\mathcal{M}(\mathcal{P} \cup \mathcal{Q}) \models \forall \overline{x} \ \forall \overline{y} \ (\Gamma(\overline{x}, \overline{y}) \leftarrow \Delta(\overline{x}), P(\overline{x}, \overline{y}))$.*

Definition 2 (Unfolding right (nfi)). *Let $\mathcal{P}$ be a program, $\pi_0 : \Gamma \leftarrow \Delta, A$ a formula and $C = \{c_1, \ldots, c_k\}$ the set of clauses of $\mathcal{P}$ such that $c_i : B_i \leftarrow \Delta_i$ and there exists a substitution $\theta_i = mgu(B_i, A)$. Then nfi[3] on π_0 w.r.t to the atom A yields a conjunction of k formulas:*

$$< \pi_0 \ : \ (\Gamma \leftarrow \Delta, A) \mid P_0 >$$
$$\downarrow \texttt{nfi}$$
$$< \pi_i \ : \ (\Gamma \leftarrow \Delta, \Delta_i)\theta_i \mid P_i >_{i=1,\ldots,k}$$

The set of definite clauses $\{P_0\theta_i \leftarrow P_i, i = 1, ..., k\}$ is generated and added to the program to be synthesized.

Example 1. Consider the formula $\pi_0 \ : \ plus(u, v, w) \leftarrow plus(v, u, w)$ and the corresponding corrective predicate is $P_0(u, v, w)$. The predicate $plus$ is defined as follows:

$$\mathcal{PLUS} \begin{cases} plus(0, x, x) & \leftarrow \\ plus(s(x), y, s(z)) & \leftarrow plus(x, y, z) \\ nat(0) & \leftarrow \\ nat(s(x)) & \leftarrow nat(x) \end{cases}$$

The application of nfi on π_0 with $\theta_1 = \{v/0, u/x, w/x\}$ and $\theta_2 = \{v/s(x),$ $u/y, w/s(z)\}$ yields the two following formulas:

$\pi_1 \ : \ plus(x, 0, x) \leftarrow \mid P_1(x)$

$\pi_2 \ : \ plus(y, s(x), s(z)) \leftarrow plus(x, y, z) \mid P_2(y, x, z)$

and the corrective clauses synthesized are:

$P_0(x, 0, x) \leftarrow P_1(x)$

$P_0(y, s(x), s(z)) \leftarrow P_2(y, x, z)$.

Next we have to synthesize the definitions of P_1 and P_2 by proving π_1 and π_2.

[3] nfi stands for *n*egation as *f*ailure *i*nference.

Definition 3 (Unfolding left (dci)). *Let $\mathcal{P}$ be a program and π the formula $\Gamma, A \leftarrow \Delta$. Suppose there exists a clause $c : B \leftarrow \Delta'$ in $\mathcal{P}$ and an existential substitutions $\theta = mgu(B, A)$. The rule of dci*[4] *applied on π w.r.t the atom A yields the singleton $\{\pi'\}$:*

$$\begin{array}{c} < \pi \ : \ (\Gamma, A \leftarrow \Delta) \mid P > \\ \downarrow \texttt{dci} \\ < \pi' = ((\Gamma, \Delta')\theta \leftarrow \Delta) \mid P' > \end{array}$$

The clause $P\theta \leftarrow P'$ is then generated.

Example 2. Consider the formula $\pi : plus(s(u), s(v), s^2(w)) \leftarrow plus(u, v, w)$ and $P(u, v, w)$ the corresponding corrective predicate. Then *dci* on π yields:

$$\pi' : plus(u, s(v), s(w)) \leftarrow plus(u, v, w) \mid P'(u, v, w)$$

and the clause $P(u, v, w) \leftarrow P'(u, v, w)$ is generated.

Definition 4 (Folding right (cutr)). *Let $\pi_1 : \Gamma \leftarrow \Delta_1, \Delta_2$ and $\pi_0 : \Lambda \leftarrow \Pi$ be two formulas satisfying the following conditions : (i) θ is a substitution such that $\Pi\theta = \Delta_1$, (ii) for any local variable x in Π, $x\theta$ is a variable and does not occur other than in $\Pi\theta$, and (iii) θ replaces different local variables in Π with different local variables in Δ_1. Then cutr (cut right) on π_1 using π_0 yields the singleton $\{\pi_2\}$:*

$$\begin{array}{c} < \pi_0 \ : \ (\Lambda \leftarrow \Pi) \mid P_0 > \\ \downarrow \\ \ldots \\ < \pi_1 \ : \ (\Gamma \leftarrow \Delta_1, \Delta_2) \mid P_1 > \\ \downarrow \texttt{cutr} \\ < \pi_2 \ : \ (\Gamma \leftarrow \Lambda\theta, \Delta_2) \mid P_2 > \end{array}$$

The clause $P_1 \leftarrow P_0\theta, P_2$ is generated and added to the set of clauses of the program to be synthesized. The rule of cutr controls the application of induction hypotheses.

Example 3. Going back to the example 1, one can remark that the right hand side of π_2 is an instance of the right hand side of π_0 with the substitution $\theta = \{v/x, u/y, w/z\}$. We can therefore apply the rule of *cutr* on π_2 using π_0, and we get the formula

$$\pi_3 : \ plus(y, s(x), s(z)) \leftarrow plus(y, x, z) \mid P_3(y, x, z)$$

and the definite clause $P_2(y, x, z) \leftarrow P_0(y, x, z), P_3(y, x, z)$ is generated.

Definition 5 (Folding left (cutl)). *Let $\pi_1 : \Gamma_1, \Gamma_2 \leftarrow \Delta$ and $\pi_0 : \Lambda \leftarrow \Pi$ be two formulas satisfying the following conditions: (i) θ is a substitution such that $\Lambda\theta = \Gamma_1$, (ii) for any local variable z in Λ, $z\theta$ is a variable and does not occur other than in $\Lambda\theta$, and (iii) θ replaces different local variables in Λ with different local variables in Γ_1. Then the application of cutl (cut left) on π_1 using π_0 yields the singleton $\{\pi_2\}$:*

$$\begin{array}{c} < \pi_0 \ : \ (\Lambda \leftarrow \Pi) \mid P_0 > \\ \downarrow \\ \ldots \\ < \pi_1 \ : \ (\Gamma_1, \Gamma_2 \leftarrow \Delta) \mid P_1 > \\ \downarrow \texttt{cutl} \\ < \pi_2 \ : \ (\Pi\theta, \Gamma_2 \leftarrow \Delta) \mid P_2 > \end{array}$$

[4] dci stands for *d*efinite *c*lause *i*nference.

The clause $P_1 \leftarrow P_0\theta, P_2$ is generated.

Definition 6 (Structural induction (indstr)). *We present the rule of structural induction for the case the variables are of type natural number. Let $\pi_0 : \Gamma(n, m, X) \leftarrow$ be a formula and $P_0(n, m, X)$ the corresponding corrective predicate. Then structural induction w.r.t n yields:*

$\pi_1 : \Gamma(0, m, X) \quad \leftarrow \qquad\qquad\qquad\quad \mid P_1(m, X)$

$\pi_2 : \Gamma(s(u), m, X) \leftarrow \Gamma(u, ?m, x) \quad \mid P_2(u, m, X, ?m, x)$

The variable m in the hypothesis of π_2 is a meta-variable *(an existential variable that occurs in the hypothesis) as it is not used for induction [1]. In the hypothesis it is written $?m$. Notice that π_2 is not an implicative formula but it will be if $?m$ is instantiated by a universal variable. The synthesized program is:*

$P_0(0, m, X) \quad \leftarrow P_1(m, X)$

$P_0(s(u), m, X) \leftarrow P_0(u, m, X), P_2(u, m, X, ?m, x)$

Example 4. Consider the formula $\pi : plus(v, U, w) \leftarrow$ associated with the corrective predicate $P_0(v, U, w)$. By structural induction w.r.t v, we obtain:

$\pi_1 : plus(0, U, w) \quad \leftarrow \qquad\qquad\qquad\quad \mid P_1(U, w)$

$\pi_2 : plus(s(x), U, w) \leftarrow plus(x, u, ?w) \quad \mid P_2(x, U, w, u, ?w)$

the predicate P_0 is defined by

$P_0(0, U, w) \quad \leftarrow P_1(U, w)$

$P_0(s(x), U, w) \leftarrow P_0(x, u, ?w), P_2(x, U, w, u, ?w)$

Definition 7 (Simplification rule (simp)). *Let $\pi : A, \Gamma \leftarrow B, \Delta$ be a formula such that there exists θ satisfying $A\theta = B$ and θ substitutes only existential variables of A. Then simp on π yields the singleton $\{\pi'\}$:*

$$< \pi \ : \ (A, \Gamma \leftarrow B, \Delta) \mid P >$$
$$\downarrow \texttt{simp}$$
$$< \pi' \ : \ (\Gamma\theta \leftarrow \Delta) \mid P' >$$

The clause $P\theta \leftarrow P'$ is then generated.

Example 5. Consider the formule

$\pi : plus(x, y, X), plus(X, z, V) \leftarrow plus(x, y, t) \quad \mid \ P(x, y, X, z, V, t)$

With the existential substitution $\theta = \{X/t\}$, π can be simplified into

$\pi' : plus(t, z, V) \leftarrow \qquad \mid P'(t, z, V)$

The clause $P(x, y, t, z, V, t) \leftarrow P'(t, z, V)$ is then generated.

Definition 8 (POSTULATE (post)). *Let $\pi : \Gamma \leftarrow$ be an implicative formula and P be a corrective predicate associated with π. Then the application of the rule of postulate on π yields the formula true and the corrective clause $P \leftarrow \Gamma$.*

$$< \Gamma \leftarrow \quad \mid P >$$
$$\downarrow \texttt{post}$$
$$< \ true \ \mid true >$$

Example 6. In the example (1) page 14, to complete the proof of π_1 we can postulate $plus(x, 0, x)$, and we obtain the corrective clause $P_1(x) \leftarrow plus(x, 0, x)$.

Proposition 1 ([6, 3]). *The rules of nfi, dci, indstr, simp and post preserve partial correctness.*

Proposition 2 ([3]). *The rules of cutr and cutl preserve partial correctness.*

Definition 9 (FAILURE (fail)). *Let $\mathcal{P}$ be a program, $\pi : \Gamma \leftarrow \Delta$ be a formula and P be a corrective predicate associated with π. If no atom of Γ is unifiable with no clause head of $\mathcal{P}$ and that $\mathcal{M}(\mathcal{P}) \models \Delta$ then the rule of failure is applied and yields the formula $false$:*

$$\begin{array}{c} < \Gamma \leftarrow \Delta \ \mid P > \\ \downarrow \texttt{fail} \\ < false \ \mid false > \end{array}$$

This rule allows us to detect totally false conjectures.

Proposition 3. *The rule of failure preserves partial correctness.*

Proof. It is easy to see that the formula $\Gamma \leftarrow \Delta, P$ holds as P is the predicate $false$.

Example 7. Suppose we have to prove the formula : $plus(s(v), U, 0) \leftarrow nat(v)$. The atom $plus(s(v), U, 0)$ is $false$ because in one hand it cannot be reduced using the program $\mathcal{PLUS}$ and on the other hand we have $\mathcal{M}(\mathcal{PLUS}) \models \forall x\ nat(x)$. The formula $plus(s(v), U, 0) \leftarrow nat(v)$ is then false and the corresponding corrective predicate is set to $false$.

3 Predicate synthesis

We define the notion of counterexemple that allows us to detect and to locate errors in computer systems. Our definition of counterexemple is similar to the definition of [13]. This automatic generation of counterexemples is an important tool in the design and debugging of systems.

Definition 10 (Conterexample of a formula). *Let $\mathcal{P}$ be a program. An example of an implicative formula $\Gamma \leftarrow \Delta$ is a substitution σ such that: (i) all the universally quantified variables in the formula are instantiated to ground terms by σ, i.e., $\Delta\sigma$ is ground, and (ii) $\mathcal{M}(\mathcal{P}) \models \Delta\sigma$.*
A counterexample is an example σ but $\mathcal{M}(\mathcal{P}) \not\models \exists(\Gamma\sigma)$.

Theorem 1 (Propagation of a counterexample [13]). *If there is a counterexample on a node N in a proof tree, there is at least one successor of N on which there is a counterexample.*

Example 8. Let us consider the formula: $plus(v, U, w) \leftarrow nat(v), nat(w)$ associated with a corrective predicate $P_0(v, U, w)$. The corresponding proof tree is depicted by the figure (1). The branch corresponding to the case v=s(x) and w=0 cannot be closed since $plus(s(x), U, 0)$ is fully false and $nat(x)$ is true. Therefore this branch suggests counterexamples of the form $\{v/s^k(x_0), w/0, k \geq 0\}$ where x_0 is a ground term. Note that each node i is associated with a corrective predicate P_i. The logic program corresponding to the figure (1) is :

(1) $P_0(0,U,w) \leftarrow P_1(U,w)$		(3) $P_1(w,w) \leftarrow$
(2) $P_0(s(v),U,w) \leftarrow P_2(v,U,w)$		(8) $P_7(v,w) \leftarrow P_8(w)$
(5) $P_2(s(v),U,s(w)) \leftarrow P_5(v,U,w)$		(9) $P_8(w) \leftarrow$
(6) $P_5(v,U,w) \leftarrow P_6(v,U,w)$		
(7) $P_6(v,U,w) \leftarrow P_0(v,U,w), P_7(v,w)$		

A straightforward unfolding process w.r.t the intermediate predicates simplifies the synthesized program as $\mathcal{Q}$:

$$P_0(0,w,w) \leftarrow$$
$$P_0(s(v),U,s(w)) \leftarrow P_0(v,U,w)$$

A truncation of $\mathcal{Q}$ w.r.t the second argument, the argument of the existential variable which is unchangeable in the recursive call of P_0, yields $\mathcal{Q}'$:

$$P'_0(0,w) \leftarrow$$
$$P'_0(s(v),s(w)) \leftarrow P'_0(v,w)$$

which is exactly the definition of the relation $\leq$, i.e. $P'_0(x,y)$ means that $x \leq y$, and we have the property of partial correctness:

$$\mathcal{M}(\mathcal{PLUS} \cup \mathcal{Q}') \models (plus(v,U,w) \leftarrow \ nat(v), nat(w), P'_0(v,w))$$

We show below that corrective predicates are helpful when we are dealing with tail recursive functions with accumulator argument. The verification of such functions requires generalization technique. To illustrate, let's take a non-trivial example.

Example 9 (Example of generalization). Suppose we want to prove the true formula:

$$qrev(x,[],U), rev(U,x) \leftarrow \tag{1}$$

where $qrev$ is the tail recursive version of rev and the predicates rev, app and $qrev$ are defined below. The proof of (1) by induction will fail as induction hypothesis cannot be applied.

$$\mathcal{R}ev \begin{cases} rev([],[]) \leftarrow \\ rev([a|x],z) \leftarrow rev(x,y), app(y,[a],z) \\ app([],x,x) \leftarrow \\ app([a|x],y,[a|z]) \leftarrow app(x,y,z) \\ qrev([],x,x) \leftarrow \\ qrev([a|x],y,z) \leftarrow qrev(x,[a|y],z) \end{cases}$$

We therefore generalize the formula as follows:

$$\varphi : qrev(x,z,U), rev(U,V) \leftarrow$$

and we have to synthesize a program that computes V in term of the universal variables x and z. However the proof of φ is hard enough. Then one can look for a formula equivalent to φ but which is easier to prove than φ. Our aim is to improve the provability of φ using corrective predicates. To do that we are looking for a maximal corrective predicate[5] P_0 that satisfies.

$$(qrev(x,z,U), rev(U,V) \leftarrow) \leftrightarrow P_0(x,z,V) \tag{2}$$

To prove the original formula it is sufficient to synthesize a computable definition of P_0 and to show if z=[] then V=x and $P_0(x,[],x)$ is true. In order to achieve this purpose

[5] P is maximal for φ iff the formula $\varphi \leftrightarrow P$ holds.

we start by structural induction on the variable x of type list and we get:

$$qrev([],z,U), rev(U,V) \quad \leftarrow \quad | \; P_1(z,V) \tag{3}$$

$$qrev([a|x],z,U) rev(U,V) \leftarrow qrev(x,?z,u) rev(u,V) | P_2(a,x,z,V,?z,u,v) \tag{4}$$

where $?z$ is a meta-variable. The corrective clauses :

$P_0([],z,V) \quad \leftrightarrow P_1(z,V)$

$P_0([a|x],z,V) \leftrightarrow P_0(x,?z,u,v), P_2(a,x,z,V,?z,u,v)$

are generated. We apply the rule of *dci* on (3) and we postulate the atom rev(z,v):

$P_0([],z,V) \quad \leftarrow P_1(z,V)$

$P_1(z,V) \quad \leftrightarrow rev(z,V)$

$P_0([a|x],z,V) \leftrightarrow P_0(x,?z,u,v), P_2(a,x,z,V,?z,u,v)$

with the existential substitutions $\{?z/[a|z], U/u, V/v\}$, (4) is simplified into true, i.e. P_2 is set to true and the synthesized program is:

$P_0([],z,V) \quad \leftrightarrow P_1(z,V)$

$P_1(z,V) \quad \leftrightarrow rev(z,V)$

$P_0([a|x],z,v) \leftrightarrow P_0(x,[a|z],u,v)$

by unfolding process the intermediate predicate P_1 is eliminated, we get:

$$P_0([],z,V) \quad \leftrightarrow rev(z,V) \tag{5}$$

$$P_0([a|x],z,v) \leftrightarrow P_0(x,[a|z],v) \tag{6}$$

that can be compiled into a Prolog program. When unfolding the right-hand-side of (6) w.r.t (5),we obtain :

$$P_0([a],z,v) \leftrightarrow rev([a|z],v)$$

by the definition of rev we get:

$$P_0([a],z,v) \leftrightarrow rev(z,u), app(u,[a],v).$$

The constant $[a]$ can now be generalized on both sides, and we get a computable definition of P_0: $\quad P(x,z,v) \leftrightarrow rev(z,u), app(u,x,v)$.

We replace in (2) P_0 by its definition and we get:

$$qrev(x,z,u), rev(u,v) \leftrightarrow rev(z,u), app(u,x,v).$$

Now if $z = []$ we have the formula:

$$qrev(x,[],U), rev(U,y) \leftrightarrow rev([],u), app(u,x,v).$$

By the definition of rev, we have: $qrev(x,[],u), rev(u,y) \leftrightarrow app([],x,v)$, and by the definition of app, we have:

$$qrev(x,[],U), rev(U,x) \leftrightarrow true$$

this completes the proof of the conjecture (1).

4 Related Works

Franŏvà et al. [5] and Protzen [12] have investigated the problem of patching faulty conjectures, but there methods are illustrated by simple examples in arithmetic and where the error can be detected in the base cases (not in the recursive calls). Also Monroy et al. have introduced a method for correcting faulty conjectures [10]. However, they only partially deal with the problem of correcting faults. For example, they cannot build a corrective predicate, only identify it as long as it is present in the working theory. Monroy has proposed in [9] another method that consists of a collection of construction

commands and is able to synthesize corrective predicates. Kapur et al. have proposed an interesting method based on corrective predicates to define classes of formulas where inductive validity is decidable [7].

5 Final Remarks

We have presented a method for patching faulty conjectures by synthesizing definite programs. The contribution of the paper is mainly the construction of corrective predicates by completing failed proof attempts and the method is integrated with the interactive theorem prover SPES [4]. Patching faulty conjectures is particularly interesting when the formulas to be proved have to be (over-)generalized. We have tested our method on several examples on natural numbers, lists and trees, see [3], and it is successfully implemented in the functional language Objective Caml. If our proof system is used to prove a faulty conjecture, it will on the fly to build a candidate corrective predicate. As illustrated in the table (1), the original conjecture is not necessary false. For example, the conjecture (18) specifies the sort algorithm by permutations and we have synthesized the algorithm of sort by insertion. We are currently trying to characterize the conjectures that can be corrected and to propose a general method for spotting incorrect generalization. Another interesting track is to show that our method may be used for finding bugs in recursive algorithms and for discovering attacks on security protocols.

References

1. R.M. Burstall. Proving properties of programs by structural induction. *Computer journal*, 12(1) 41-48, 1969.
2. R.M. Burstall and J.A Darlington. Transformation System for Developing Recursive Programs. *Journal of the Association for Computing Machinery*, 24(1) 44-67, 1977.
3. A. Francis, K. Bsaïes and M. Demba. *Predicate synthesis from inductive proof attempt of faulty conjectures.* In M. Bruynooghe, editor *Proc. of the 13th International Symposium on Logic-based Program Synthesis and Transformation LOPSTR'03. Uppsala, Sweden, August 25-27, 2003. Revised selected papers. LNCS 3018*, Springer-Verlag, 2004.
4. A. Francis, K. Bsaïes, J.P. Finance and A. Quéré, A. *SPES: A System for Logic Program Transformation.* In A. Voronkov *Proc. of the International Conference on Logic Programming and Automated Reasoning LPAR'92*, volume 624 of LNAI. St. Petersburg 1992.
5. M. Frănová and Y. Kodratoff. Predicate synthesis from formal specifications. In B. Neumann, editor, *proceedings of the 10th European Conference on Artificial Intelligence ECAI'92*, pages 87–91, England, 1992.
6. L. Fribourg. Extracting Logic Programs from Proofs that Use Extended Prolog Execution and Induction. In J.M. Jaquet, editor *Constructing Logic Programs, Chapter 2, pages 39–66, Wiley, 1993.*
7. J. Giesl and D. Kapur. Decidable classes of inductive theorems. In *IJCAR'01, First International Joint Conference on Automated Reasoning*, Italy, 2001.
8. A. Kakas, R.A. Kowalski, and F. Toni. *Handbook of logic in Artificial Intelligence and Logic Programming*, volume 5, chapter The Role of Abduction in Logic Programming, pages 235–324. Oxford University Press, 1998.
9. R. Monroy. The use of Abduction and Recursion-Editor Techniques for the Correction of Faulty Conjectures. In *Automated Software Engineering*, 2000.
10. R. Monroy, A. Bundy, and A. Ireland. Proof plan for the correction of false conjectures. In F. Pfenning, editor, *Proceedings of the 5th Int. Conf. on Logic Programming and Automated Reasoning, LPAR'94*, volume 822 of *LNAI*, pages 54–64, Kiev, Ukraine, Springer-Verlag, 1994.

N^o	Conjectures	Corrective predicate	Definitions
1	$flip(x,Y), flip(Y,Z) \leftarrow$	P(x,Z)	$P(leaf(x), leaf(x)) \leftarrow$ $P(branch(x,y), branch(v,w))$ $\leftarrow P(x,v), P(y,w)$
2	$qrev(x,z,X), qrev(X,[],t) \leftarrow$	P(x,z,t)	$P(x,z,t) \leftrightarrow qrev(z,x,t)$
3	$half(y,s(0)) \leftarrow$	P(y)	$y = s^2(0)$; $y = s^3(0)$
4	$half(x,Y), double(Y,Z) \leftarrow$	P(x)	$P(0) \leftarrow$ $P(s^2(x)) \leftarrow P(x)$
5	$sort(l,l) \leftarrow list(l)$	P(l)	$P([]) \leftarrow$ $P([a]) \leftarrow$ $P([a,b\|l]) \leftarrow inf(a,b), P([b\|l])$
6	$even(z) \leftarrow plus(x,y,z)$	P(x,y,z)	$P(0,y,y) \leftarrow even(y)$ $P(s(0),y,s(y)) \leftarrow odd(y)$ $P(s^2(x),y,s^2(z)) \leftarrow P(x,y,z)$
7	$even(n) \leftarrow len(x,n)$	P(x,n)	$P([],0) \leftarrow$ $P([a,b\|x], s^2(n)) \leftarrow P(x,n)$
8	$insert(a,l,[a\|l]) \leftarrow list(l)$	P(a,l)	$P(a,[]) \leftarrow$ $P(a,[b\|l]) \leftarrow P(a,l), inf(a,b)$
9	$(x-y)+z = (x+z)-y$	P(x,y)	$P(x,0) \leftarrow$ $P(s(x),s(y)) \leftarrow P(x,y)$
10	$app(u,v,X), len(X,N) \leftarrow$list(u)	P(u,v,N)	$P([],v,N) \leftarrow len(v,N)$ $P([a\|u],v,s(N)) \leftarrow P(u,v,N)$
11	$ord(v) \leftarrow ord(u), place(a,u,v)$	P(a,u,v)	$P(a,[],[a]) \leftarrow$ $P(a,[b\|u],[a,b\|u]) \leftarrow a \leq b$ $P(a,[b\|u],[b\|v]) \leftarrow P(a,u,v), b \leq a$
12	$ord(U), insert(a,U,y) \leftarrow ord(y)$	P(a,y)	$P(a,[a]) \leftarrow$ $P(a,[b\|y]) \leftarrow P(a,y)$
13	$place(a,u,V), ord(V) \leftarrow ord(u)$	P(a,u,V)	$P(a,[],[a]) \leftarrow$ $P(a,[b\|u],[a,b\|u]) \leftarrow lessthan(a,[b\|u])$ $P(a,[b\|u],[b\|V]) \leftarrow P(a,u,V)$ $lessthan(b,V)$
14	$qrev(x,[],T), rev(x,T) \leftarrow$	P(x,T)	is generalized to the conjecture (16)
15	$qrev(x,[],X), qrev(X,[],x) \leftarrow$	P(x,X)	is generalized to the conjecture (2)
16	$qrev(x,z,T), rev(t,T) \leftarrow$	P(x,z,t)	$P(x,z,t) \leftrightarrow rev(z,u), app(u,x,t)$
17	$mult(U,v,w) \leftarrow nat(v),\ nat(w)$	$P(w,v)$	$P(0,0) \leftarrow$. $P(0,s(u)) \leftarrow$ $P(s(u),v) \leftarrow Q(u,v,w), P(w,v)$ $Q(0,0,s(0)) \leftarrow$ $Q(s(u),v,s(w)) \leftarrow Q(u,v,w)$
18	$perm(x,Y), ord(Y) \leftarrow list(x)$	P(x,Y)	$P([],[]) \leftarrow$ $P([a\|x],Y) \leftarrow P(x,y), Q(a,y,Y)$ $Q(a,[],[a]) \leftarrow$ $Q(a,[b\|t],[a,b\|t]) \leftarrow inf(a,b)$ $Q(a,[b\|y],[b\|t]) \leftarrow inf(b,a), Q(a,y,t)$
19	$even(z) \leftarrow plus(x,x,z)$	P(x,z)	is generalized to the conjecture (6)

Table 1. Patched conjectures and programs extracted.

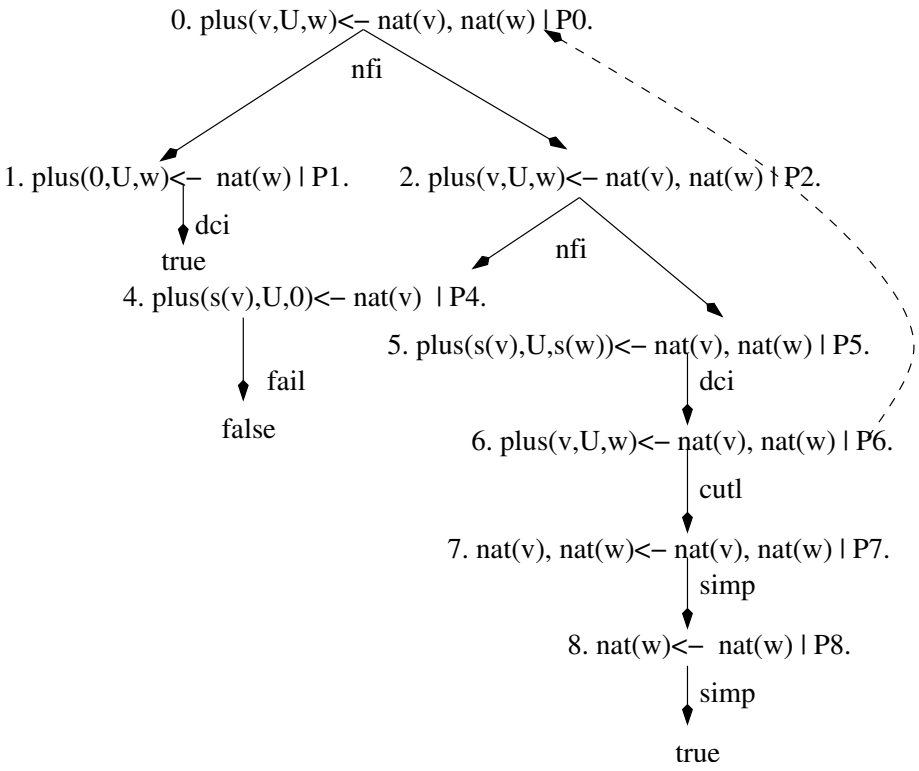


Fig. 1. Proof tree of plus(v,U,w)← nat(v), nat(w).

11. C. S. Peirce. Collected Papers of Charles Sanders Peirce. C. Harston and P. Weiss. editors, Harvard University Press, 1959.
12. M. Protzen. Patching faulty conjectures. In M. McRobbie and Slaney, editors, *Proceedings of the 13th Int. Conf. on Automated Deduction, CADE13*, volume 1104 of LNAI, pages 77–91, New Brunswick, NJ,USA, 1996.
13. A. Sakurai and H. Motoda. Proving Definite Clauses without Explicit Use of Inductions. In K. Furukawa, H. Tanaka, and T. Fujisaki, editors, *Proceedings of the 7th Conference, Logic Programming'88*, volume 383 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1988.

Disproving Distributivity in Lattices Using Geometric Logic*

Marc Bezem†

Abstract

We report on experiments finding countermodels with a theorem prover for geometric logic. The main insight we acquire here is that it can be necessary to *strengthen* the geometric theory in order to find certain models. More precisely, we have added to the the theory of lattices an equality theory which enforces the system to consider quotients of the syntactic models. In this way our system recovered the two minimal 5-point lattices which are not distributive. One of these is modular and the other is not. Disproving distributivity in modular lattices has been the most demanding test so far. We discuss the results and some variations on the equality theme.

1 Introduction

Geometric logic, also called coherent logic, originated in algebraic geometry, see for example [6, Sect. 16.4]. In this note we will work in the first-order fragment of geometric logic (abbreviated by GL) consisting of (implicitly) universally quantified implications of the following form:

$$A_1 \wedge \cdots \wedge A_n \to E_1 \vee \cdots \vee E_m$$

Here the A_i are first-order atoms. In contrast to resolution logic [11], where the E_j must also be atoms, they may here be existentially quantified conjunctions of atoms. Thus the general format of a *geometric formula* reads:

$$A_1 \wedge \cdots \wedge A_n \to \exists \vec{x}_1.C_1 \vee \cdots \vee \exists \vec{x}_m.C_m \tag{1}$$

where the C_j are conjunctions of atoms. The special cases $n = 0$, $m = 0$ and no existential quantification, in all possible combinations, are understood to be included. (If the premiss is empty we leave out the $\to$ as well.) A *geometric theory* is a set of geometric formulas. Closed atoms will also be called *facts*.

What then are the virtues of GL, given the fact that the existential quantifiers could easily be skolemized after which the formula becomes equivalent to a set of clauses? First of all, skolemization changes the meaning of the formula. The skolemized formula is stronger than the original formula and equivalence can only be obtained by postulating

*Please note that this is work in progress.

†Department of Computer Science, Bergen University, P.O. Box 7800, N-5020 Bergen, Norway, bezem@ii.uib.no

- weak instances of the Axiom of Choice, called Skolem axioms, and
- for the constructivist, instances of the Independence of Premiss Axiom:

$$(\forall x.\ (\phi(x) \rightarrow \exists y.\ \psi(x,y))) \ \rightarrow\ \forall x \exists y.\ (\phi(x) \rightarrow \psi(x,y))$$

Thus your reasoning assistant works on a different problem than you! This is unfortunate if you are interested in constructive proofs and not only in classical truth. It may also be difficult to guide your assistant.

GL has a natural proof theory from which proof objects easily can be obtained. Reasoning in GL is constructive and can be used for, e.g., the constructivization of classical abstract algebra, see [5]. The proof theory is sound and complete with respect to Tarskian semantics. As a consequence, classical logic is conservative over GL. (GL shares this virtue with resolution logic, but the fragment is much larger.) This may even palliate the constructivist's scruples with respect to the abovementioned use of Choice and Independence of Premiss in the following ironic way. Having proved a geometrical consequence of a geometrical theory *in whatever non-constructive way* one can claim this result to be constructive. A constructivist may then mistrust the methods, but has to accept the result (and has to work for his own proof).

A substantial number of reasoning problems (e.g., in confluence theory, lattice theory and projective geometry[1]) can be formulated *directly* in GL without any clausification or skolemization. This gives some additional benefits in terms of guiding an automated theorem prover and using the proof objects in other logical frameworks. In [1] the automation of GL has been studied, inspired by the system SATCHMO [8] for resolution logic.

There are several other reasons why geometric logic is interesting. See [2] for its relevance to computer science and [10] for one example of its interesting proof-theoretic properties.

2 Proof system

GL has a natural proof system which is based on forward (ground) reasoning with case distinction. Existential quantifiers are eliminated by introducing witnesses. In order to explain this a bit more, let T be a geometric theory. Assume we have a domain I of initial constants and *witnesses*, constants introduced during the reasoning process. Let X be a set of facts in which only constants from I occur. Together I and X form a so-called (reasoning) *state*. A reasoning *step* in this state consists of picking a closed I-instance $C \rightarrow D$ of an axiom from T that is invalid in the state. This means that the premiss C is true in the state, but the conclusion D is not. More precisely, this means that all facts in C occur in X, but for no disjunct $\exists \vec{x}.C_j$ of D there exist witnesses $\vec{w}$ such that $C_j[\vec{x}{:=}\vec{w}]$ is true in X.

As an example, consider a state with $I = \{0\}$, $X = \{Nat(0)\}$ and an axiom

$$Nat(x) \rightarrow \exists y.(Nat(y) \wedge S(x,y)) \tag{2}$$

[1] However, the qualifier 'geometric' does not come from projective geometry, but refers to the origin of GL in algebraic geometry.

The instance of this axiom with x:=0 is invalid in the state as the premiss is true but the conclusion is not. The reasoning step now so to say remedies this failure by making the conclusion of the instance true by adding a witness to I, suggestively denoted as 1, and adding the facts $Nat(1)$ and $S(0,1)$ to X. The reasoning process would then continue in the state with $I' = \{0,1\}$ and $X' = \{Nat(0), Nat(1), S(0,1)\}$. Assume we would like to prove $G = \exists xy.(S(0,x) \wedge S(x,y))$. Then we would pick in the new state the instance with x:=1 of the above axiom to arrive in a state with constants $I'' = \{0,1,2\}$ and facts $X'' = \{Nat(0), Nat(1), S(0,1), Nat(2), S(1,2)\}$. Now we can stop since the goal G is true in this state: just take x:=1 and y:=2 and observe that $S(0,1), S(1,2) \in X''$. Note that the suggestive names for the witnesses are inessential. It is, however, important to avoid name conflicts.

In the case of a disjunctive conclusion of the axiom the reasoning process forks and the goal has to be proved in all the branches corresponding to the disjuncts in the conclusion. In the special case of an empty disjunction there are no branches and we are done. This special case corresponds to the Ex Falso rule.

The above procedure actually proves the geometric formula

$$Nat(x) \to \exists yz.(S(x,y) \wedge S(y,z))$$

from the axiom (2), since nothing special has been assumed about the constant 0. In this way the procedure easily generalizes to arbitrary geometric formulas and geometric theories. The resulting proof system is sound and complete with respect to Tarskian truth, see for example [1]. Take care that $\vdash, \Vdash$ in [1] look like *backward* reasoning because of their inductive definitions. However, reasoning goes actually *forward*, since the activity occurs on the left (contravariance).

Of course it is of crucial importance which instance of which axiom is applied. The consequence relation $\Vdash$ applies all (finitely many) invalid instances of all axioms simultaneously. This is called the *breadth-first* strategy. The consequence relation $\vdash$ applies only one instance at a time. We have implemented $\vdash$ with the *depth-first* (Prolog) strategy where the first invalid instance is applied.

In this note we will explore the following property of the proof system. If in some branch the procedure terminates with all axioms true but the goal still false, then the state in question constitutes a countermodel. This property holds for both the breadth-first and the depth-first variant. We only consider the latter, since we have not implemented the former. When used in theorem provers, breadth-first is in many cases less efficient than depth-first. This is not so clear in the case of disproving, but neither of the two strategies satisfies *negative* completeness, the property that a countermodel is found whenever the goal does *not* follow from the theory. The reason is that only finite countermodels are found, whereas in some cases there are only infinite countermodels. However, even if there exist finite countermodels we do not always find them, but exactly here we will show how to improve the situation by extending the theory. Finding finite countermodels with Extended Positive Tableaux has been studied in [3].

3 Lattices

A *lattice* is a partial ordering equipped with two binary operations called *join* and *meet* yielding the least upper bound and the greatest lower bound, respectively. As

GL is currently implemented without function symbols we formalize lattice theory with ternary predicates `j` and `m` for join and meet, respectively, besides a binary predicate `lt` for the ordering. The following axioms are taken from the actual input file, but we have left out those parts that are not relevant for disproving, notably the annotations for reconstructing proofs and for enforcing certain strategies.

```
dom(X) => lt(X,X).                          % reflexivity
lt(X,Y),lt(Y,Z) => lt(X,Z)).                % transitivity
m(X,Y,Z) => lt(Z,X),lt(Z,Y)).               % Z lower bound of X,Y,
m(X,Y,Z),lt(U,X),lt(U,Y) => lt(U,Z)).       % even the greatest
j(X,Y,Z) => lt(X,Z),lt(Y,Z)).               % Z upper bound of X,Y,
j(X,Y,Z),lt(X,U),lt(Y,U) => lt(Z,U)).       % even the smallest
lt(X,Y) => m(X,Y,X),j(X,Y,Y)).              % minimum and maximum
m(X,Y,Z) => m(Y,X,Z)).                      % commutativity of m
j(X,Y,Z) => j(Y,X,Z)).                      % commutativity of j
m(X,Y,U),m(U,Z,V),m(Y,Z,W) => m(X,W,V)).    % associativity of m
j(X,Y,U),j(U,Z,V),j(Y,Z,W) => j(X,W,V)).    % associativity of j
%equality theory to be inserted here
dom(X),dom(Y) => dom(U),m(X,Y,U)).          % existence of meets
dom(X),dom(Y) => dom(U),j(X,Y,U)).          % existence of joins
```

A few remarks are in order here. In addition to ordinary Prolog syntax we use a domain predicate `dom` for achieving *range restriction*, the property that every (implictly) universally quantified variable occurs on the left-hand side of the implication `=>`. An example is the first axiom stating reflexivity. If `dom` occurs on the right then it means existential quantification. The last two axioms stating the existence of meets and joins, respectively, show both uses of `dom`.

One may miss axioms ensuring that the join and meet relations are functional.[2] In fact the ordering is just a preorder since we have not required antisymmetry. Clearly, `j(a,b,c)` and `j(a,b,d)` imply both `lt(c,d)` and `lt(d,c)` without `c` and `d` being necessarily equal. However, such `c` and `d` may be taken to be equivalent and this equivalence is a congruence with respect to all predicates involved. Any model can thus be collapsed into a lattice. The axioms on commutativity and the minimum and the maximum can be understood as redundant but convenient.

Given the fact that the prover uses a depth-first strategy, the order of the axioms is important. The typical order is: first the Horn clauses, then clauses with disjunction (absent above, but present in the equality theory to be inserted later) and finally clauses with existential quantifiers.

Distributivity of the join over the meet normally reads

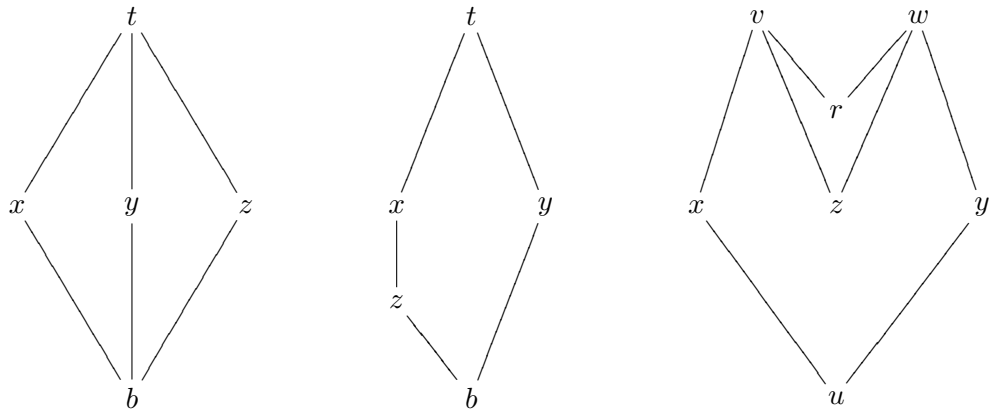
$$(x \cap y) \cup z = (x \cup z) \cap (y \cup z)$$

and can be formalized using ternary predicates as follows:

```
m(X,Y,U),j(X,Z,V),j(Y,Z,W),m(V,W,R) => j(U,Z,R).
```

[2] Actually, the proof system is such that at most one join will be inferred, after which the instance of the axiom has become true. Similarly for the meet axiom. Completeness is not at stake here.

Distributivity is known to be independent of the other lattice axioms and there exist two minimal countermodels up to isomorphism, graphically depicted as (i) and (ii) in the following figure.



(i) (ii) (iii)

Proving distributivity in GL from the other axioms could be attempted by adding the following axioms to the above theory

```
true => m(x,y,u),j(x,z,v),j(y,z,w),m(v,w,r).
j(u,z,r) => goal.
```

The first axiom postulates a constellation of 7 constants `x,y,z,u,v,w,r` depicted in figure (iii) above. Distributivity boils down to `j(u,z,r)`, so it would suffice to prove `goal` from the extended theory.

As distributivity is independent, one cannot expect to find a proof. However, one could hope that the prover would stop and that a countermodel can be read off from the state. This hope is idle. The prover doesn't give up constructing new meets applying the last but one axiom. By the depth-first strategy the last axiom is never applied. Interchanging the two last axioms doesn't help. What could help is applying some fair strategy for the last two axioms. Then the prover would conceivably remain busy with building an infinite countermodel against distributivity. But it would not stop.

The fundamental problem here is that the prover only considers syntactic models, a kind of minimal Herbrand models for geometric logic, which in the case of lattice theory all happen to be infinite . The finite lattices, which are quotients of syntactic models, are completely ignored. Systems like Paradox [4] and Mace [9] explore the finite models.

In this note we wish to explore another possibility, namely by strengthening the theory with an equality theory. This has the advantage that one can stay within the same framework of GL and use the same prover. The idea is to add a decidable equality to the theory in such a way that, before generating new elements, the prover considers all possible identifications of existing elements. By including proper congruence axioms in the equality theory, actually quotient models are constructed. In the case of lattice theory the following extension is a natural first attempt.

```
dom(X) => eq(X,X).                      % reflexivity of eq
eq(X,Y) => eq(Y,X)).                    % symmetry
eq(X,Y),eq(Y,Z) => eq(X,Z)).            % transitivity
eq(X,Y) => lt(X,Y),lt(Y,X)).            % congruence wrt lt
eq(X,Y),neq(X,Y) => false).             % mutual exclusion
dom(X),dom(Y) => eq(X,Y);neq(X,Y)).     % decidability of eq
```

Again a few remarks are in order. The connective `;` is Prolog's disjunction, so that the last two axioms express that `neq` is the negation of `eq` and that equality is decidable. Concerning the order of the axioms of the extended theory, the most important point is that the typical order (first Horn clauses, then general clauses and finally axioms with existential quantifiers) has been maintained by inserting the equality theory just before the existential axioms in the theory of lattices. Later we come back to the optimal placement of the axiom for mutual exclusion.

4 Results and variations

The extended theory from the previous section has been used as input to a depth-first GL prover. After a considerable run a model was found leaving `goal` unproven. Relating the state in which the prover halted with the figure (*iii*) above, it turned out that the system had generated three new constants: `C0` the meet of `x` and `z`, `C1` the meet of `y` and `z` and `C2` the join of `v` and `w`. Moreover the following identifications were made:

```
u = C0 = C1, v = w = r = C2
```

Together with `x,y,z` these form indeed the 5-point lattice (*i*). The generation of the constants could somehow be expected: since the meet axiom comes before the join axiom, the 'missing meets' are generated first, and then the 'missing join', where 'missing' depends on the identifications.

It can then also be expected that interchanging the meet and the join axiom may affect the model found. This is indeed the case: with these two axioms interchanged the only new constant generated is `C0` the join of `z` and `u`. Moreover the following identifications were made:

```
z = C0 , x = v = r
```

Consequently we have `lt(u,z)` and `lt(z,x)`, so actually the 5-point lattice (*ii*).

Several variations on this theme are possible. A weaker form of distributivity is called *modularity*. The modularity axiom normally reads

$$z \leq x \rightarrow (x \cap y) \cup z = x \cap (y \cup z)$$

and can be formalized using ternary predicates as follows:

```
lt(Z,X),m(X,Y,U),j(U,Z,V),j(Y,Z,W) => m(X,W,V).
```

Modularity follows from distributivity since $(x \cap y) \cup z = (x \cup z) \cap (y \cup z) = x \cap (y \cup z)$ if $x \cup z = x$, that is, if $z \leq x$. Also modularity is known to be independent of the other lattice axioms. Of the two minimal countermodels against distributivity, the one

depicted in (i) is modular but (ii) is not. This countermodel is easily found (in less than 10 CPU seconds), where the order of join axiom and the meet axiom doesn't make a significant difference. One may ask: why is it easier to disprove a weaker axiom? We have no good answer to this other than that in the case of modularity one constant less is involved (`r`).

As could be expected, the most demanding test is disproving distributivity in modular lattices. Here the order of the last two axioms was critical: first the meet axiom, then the join axiom. With the latter axiom preceeding the former no model was found and the testing platform became in the end unstable.

Critical readers will have observed that there are many possible variations of the equality theory. Why, for example, only congruence with respect to the ordering relation `lt`? It turned out that adding more congruence axioms in many cases slowed down the search. On the other hand, we can exploit the special property of lattices that equality is definable in terms of the ordering. Instead of the equality theory one could consider one single axiom:

```
dom(X),dom(Y) => lt(X,Y),lt(Y,X);neq(X,Y).
```

This was sufficient to disprove distributivity in the theory without the modularity axiom. Some of the countermodels were non-standard in the sense that the mutual exclusion axiom was *not* satisfied:

```
lt(X,Y),lt(Y,X),neq(X,Y) => false.
```

Note that it is perfectly sound to conclude from such non-standard countermodels that distributivity cannot be proved in the theory of lattices. Including the mutual exclusion axiom did in particular help when this axiom was placed at the very begining of the extended theory. As a general rule, axioms `... => false` should precede all other Horn clauses, since this avoids extending a state before finding it inconsistent. Then even in the case of modular lattices countermodels against distributivity are found in a couple of minutes.

The way in which strengthening the equality theory influences the search for countermodels is currently ill-understood. Less models can also mean less countermodels. Readers willing to independently verify these experiments can contact the author for the prover, the input files and some guidance on how to use them.

5 Conclusions

Admittedly, the results described here are not very impressive and require a considerable amount of human interaction. Specialized model generators like MACE [9] and Paradox [4] can find 5-point lattices almost instantaneously. For these and many other problems, MACE and Paradox are simply superior to GL. However, there exist also simple problems on which GL outperforms these systems. Consider booleans $0, 1$ as a subset of a larger domain and define equality of booleans in the obvious way:

$$E(0,0),\ E(1,1),\ \neg E(0,1),\ \neg E(0,1)$$

Assume we are interested in a relation which relates every n-tuple of booleans to a unique domain element. This can be achieved by the geometric axioms

$$R(x_1, \ldots, x_n, z) \wedge R(y_1, \ldots, y_n, z) \rightarrow E(x_1, y_1) \wedge \cdots \wedge E(x_n, y_n)$$

$$E(x_1, x_1) \wedge \cdots \wedge E(x_n, x_n) \rightarrow \exists z.\ R(x_1, \ldots, x_n, z)$$

This obviously requires at least 2^n elements in the domain. MACE had to give up the model search with $n = 3$, Paradox with $n = 4$. GL finds a model with $n = 5$ still instantaneously. The explanation is that MACE and Paradox try to find a model with a minimal domain, from size 1 onwards. There are very many structures with domains smaller than 2^n. (When asked for a model of size 2^n, Paradox promptly returns one.) GL works almost the opposite way: it generates immediately 2^n new and hence *different* constants and verifies that the other axiom holds. So the same eagerness of GL that had to be restrained in the previous section, helps to quickly find a solution here.

References

[1] M.A. Bezem and T. Coquand. Automating Geometric Logic. *Report 33/04, Research Group on Mathematical Linguistics*, Universitat Rovira i Virgili, Tarragona.

[2] A. Blass. Topoi and computation, *Bulletin of the EATCS* **36**:57–65, 1998.

[3] F. Bry and S. Torge. *Model generation for applications – A tableau method complete for finite satisfiability.* Research Report PMS-FB-1997-15, LMU, 1997.

[4] K. Claessen. Paradox, `www.cs.chalmers.se/~koen/paradox/`

[5] M. Coste, H. Lombardi, and M.F. Roy. Dynamical methods in algebra: effective Nullstellensätze. *Annals of Pure and Applied Logic* **111**(3):203–256, 2001.

[6] R. Goldblatt. *Topoi : the categorial analysis of logic.* Revised edition, North-Holland, 1984.

[7] A. Horn. On sentences which are true of direct unions of algebras, *Journal of Symbolic Logic* **16**(1):14–21, 1951.

[8] R. Manthey and F. Bry. SATCHMO: a theorem prover implemented in Prolog. In E. Lusk and R. Overbeek, editors, *Proceedings of the 9-th Conference on Automated Deduction*, Lecture Notes in Computer Science **310**:415–434, Springer-Verlag, 1988.

[9] W. McCune. Models And Counter Examples, `www-unix.mcs.anl.gov/AR/mace/`

[10] S. Negri. Contraction-free sequent calculi for geometric theories, with an application to Barr's Theorem, *Archive for Mathematical Logic* **42**:389–401, 2003.

[11] J.A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle, *Journal of the ACM* 12(1): 23–41, 1965.

Generating Counterexamples for Java Dynamic Logic

Philipp Rümmer*

July 1, 2005

Abstract

First-Order Dynamic Logic is an extension of First-Order Predicate Logic that enables propositions about programs to be made in a natural way. The adherence of a program to certain properties—like preserving invariants or compliance with pre-/postconditions—can be translated into the statement that a particular formula of Dynamic Logic is valid, which can be proved mechanically using calculi. Accordingly, dealing with programs that violate a formal specification leads to the investigation of invalid formulas. We consider a dynamic logic for Java and describe an approach for proving formulas *invalid* that works by reduction to the validity problem. Furthermore, the method allows us to derive concrete counterexamples for validity, which could be a useful tool for debugging programs or specifications.

1 Introduction

In the area of first-order predicate logic, in most cases deduction is about showing the validity of formulas, and consequently the employed calculi are mostly optimised for the positive case of formulas that *are* in fact valid. When dealing with statements about programs—with formulas that are valid if a program has certain properties—the more common case are erroneous programs and thus invalid formulas, however. Though it is often possible to gain information about the nature of a bug by examining failed proof attempts (of validity) of the invalid formula, the calculi involved are not tailored to this purpose. In general attempts of proving invalid formulas will not even terminate.

This paper is concerned with directly proving that formulas of a dynamic logic for the Java programming language [Bec01] are *invalid*. Program states are in this logic modelled as algebras over first-order vocabularies, which means that proving formulas invalid primarily is a search for algebra operations invalidating the formula. This is in general a higher-order problem and more difficult than proving formulas valid.

For the special case of algebras modelling program states, however, one can make use of the fact that programs can in finitely many execution steps only access finitely many data locations. When only searching for those invalidating states that can really be reached from some well-defined initial state—*finite variants* of the initial state—the problem becomes significantly easier. In this setting algebra operations can be

*Department of Computer Science and Engineering, Chalmers University of Technology and Göteborg University, SE-412 96 Göteborg, Sweden, philipp@cs.chalmers.se

represented using algebraic datatypes, which in turn can be defined in first-order logic with arithmetic.

Hence, proving a formula invalid can for a certain fragment of the dynamic logic be reduced to proving formulas (of the same logic) valid. From such a proof one can also derive concrete counterexamples for the original formula. When dealing with formulas assuring the correctness of a program, the counterexample shows a state in which the program behaves inconsistently with its specification. Since automation of the method appears to be possible to a large degree, this might result in a practical tool supporting software development. By employing an existing proof system for validity, the KeY system [ABB+05], we can also cover most parts of Java and handle real-world applications.

The following sections make up this paper: Sect. 2 gives a background about validity and satisfiability of formulas. Sect. 3 introduces the considered dynamic logic for Java. Sect. 4 describes the approach for disproving formulas and for searching for counterexamples. The method is illustrated with an example in Sect. 5, and conclusions and related work are given in Sect. 6, 7.

2 Background: Validity and Satisfiability Problems (Classical Logics)

Examining whether a formula is satisfiable (whether its negation is invalid) is usually regarded a more difficult problem than considering its validity. This is supported by the well-known fact that validity in first-order predicate logic (FOL) is semi-decidable and not decidable [Fit96]. In general, the truth of the statement is mostly determined by the underlying notion of semantic *structures* that govern the evaluation of formulas.[1] Given a class $\mathcal{S}$ of such structures, (i) a formula is called *valid* (in $\mathcal{S}$) if and only if it is evaluated to 'true' for all elements $S \in \mathcal{S}$, and (ii) a formula is called *satisfiable* (in $\mathcal{S}$) if there is an $S \in \mathcal{S}$ such that the formula is evaluated to 'true'.

What a structure actually is varies with the kind of the considered logic: for FOL, structures are algebras over first-order vocabularies (i.e. a universe of individuals that is equipped with operations according to the vocabulary), whereas modal logics or dynamic logics are usually given semantics in terms of *Kripke structures* [HKT00].

2.1 Validity and Satisfiability for First-Order Logic

In theorem proving, most commonly the validity of FOL formulas is examined in the context of entirely loose semantics, i.e. $\mathcal{S}$ is a class $\mathcal{S}_{\mathrm{L}}$ of all algebras over a certain vocabulary. The satisfiability of formulas is in this framework a difficult problem (for sufficiently rich vocabularies not semi-decidable) because it involves proving the existence of suitable algebra operations. A restriction to poor vocabularies—that for instance only contain nullary symbols—or to other decidable fragments of FOL in many cases diminishes the expressiveness of the logic unacceptably.

On the contrary, when using FOL as a language for program assertions or specifications, its expressiveness is often augmented by restricting $\mathcal{S}$ to a class $\mathcal{S}_{\mathrm{Pe}}$ of *arith-*

[1] The common distinction between structures and variable assignments is left out in this document. Valuations of variables can always be regarded as parts of structures.

metic structures, which are structures whose universe encloses the natural numbers and that provide the standard operations $0, 1, +, \cdot, \leq, \doteq$ [HKT00]. Particularly interesting for this paper is the special case of formulas that are *exclusively* build from the symbols $0, 1, +, \cdot, \leq, \doteq$, classical propositional junctors and first-order $\forall$, $\exists$-quantification. If closed formulas of this kind are evaluated in the singleton set $\mathcal{S}^0_{\mathrm{Pe}}$ that consists only of the algebra of natural numbers itself, then validity and satisfiability are the same property, i.e. a formula is valid if and only if it is satisfiable. At the same time the logic is theoretically more expressive than FOL with loose semantics. The present document is motivated by this observation and tries to carry over the result to a dynamic logic for the Java programming language [GJSB00]: to prove formulas of the dynamic logic invalid, they are first transformed in a way such that validity and satisfiability become equivalent.

3 Java Dynamic Logic (JavaDL)

First-order dynamic logic (DL) [HKT00] is a multi-modal extension of FOL in which modal operators are labelled with programs. There are primarily two kinds of modal operators that are dual to each other: a diamond formula $\langle p \rangle \varphi$ expresses that φ holds in at least one final state of program p. Box formulas can be regarded as abbreviations $[p]\varphi \equiv \neg\langle p \rangle \neg\varphi$ as usual. The DL formula that is probably shown most often is $\varphi \rightarrow \langle p \rangle \psi$ and states, for a deterministic program p, the total correctness of p concerning a precondition φ and a postcondition ψ.

In the sequel, p is a piece of sequential Java code, i.e. a list of Java statements without static directives like class declarations, and hence the logic is called JavaDL [Bec01].

In order to define a formal semantics of the modal operators it is necessary to capture Java memory states as structures (here first-order structures, i.e. algebras are used) over which the formulas φ, ψ can be evaluated. Java programs are then given semantics as partial functions of these structures, i.e. as functions that map pre-state structures to post-state structures.

3.1 Modelling Java States as Algebras: JavaDL Vocabularies

Reasoning in JavaDL always takes place in the context of a system of Java classes, which is supposed to be free of compile-time-errors, i.e. statically correct. In the whole document we assume that such a system is fixed. The context information is employed to derive a multi-sorted first-order vocabulary such that corresponding structures describe Java memory contents. A detailed description of this derivation can be found in [Bec01]; the following two sections summarise the aspects important for this paper.

3.1.1 JavaDL Vocabularies: Sorts

JavaDL is based on a typed first-order logic with subsorts that resemble the type hierarchies of Java. For each declared Java class C the vocabulary contains a sort of the same name. The interpretation of such a sort is later supposed to be a copy of the set $\mathbb{N}$ of natural numbers, i.e. objects of C are identified with numbers. This set represents a reservoir containing both those objects that are already instantiated and those that

can possibly be created later by a program: JavaDL uses a constant-domain semantics in which modal operators never change the domains of existing individuals.

Primitive types are covered by distinguished sorts *int*, *boolean*,[2] etc. The sort *int* is used to represent all different integer types of Java with finite range (a full account thereof is [Sch02]), and is semantically considered as the infinite set $\mathbb{Z}$ of mathematical integers. Accordingly, the interpretations of the other sorts are also fixed to appropriate sets. For technical reasons, a sort *nat* that represents natural numbers is declared.

3.1.2 JavaDL Vocabularies: Functions

The sorts *int*, *boolean*, etc. for primitive Java types are equipped with common operations, in particular $0, 1, +, \cdot, \leq, \doteq$ are defined for the integers, and there are constants `TRUE`, `FALSE` denoting the two boolean values.

In order to enumerate the particular objects of a class C, we introduce function symbols $obj_C : nat \to C$ that are interpreted with bijections. Instance attributes

```
class C { ... T a; ... }
```

are translated to unary function symbols $a : C \to T$ of the same name. Semantically, such a function can be regarded as an infinite array that contains one entry for each object of C. For distinguishing instantiated objects from non-instantiated ones, an implicit attribute `boolean <created>` is added to each class. Altogether, the instance attribute vectors model the heap during program execution.

Local variables (and also class attributes) can simply be modelled by declaring constants (nullary function symbols).

3.2 Semantics of Java Dynamic Logic

Semantics of modal logics is usually based on the notion of *Kripke structures*: structures provide a set of worlds and transition relations between these worlds. While this approach can be used to give meaning to JavaDL formulas, following it would be unnecessarily complicated because the effect of a particular Java program is completely determined by the initial values of variables and attributes. Given a collection of worlds, there would be at most one compatible transition relation, which enables us to use a simpler definition.

The set of all multi-sorted algebras over the vocabulary built in Sect. 3.1 is in the sequel denoted with $\mathcal{S}_{\mathrm{DL}}$. Interpretations of the particular sorts modelling Java are in $\mathcal{S}_{\mathrm{DL}}$ chosen to be constant and as already described, for instance a sort C representing a Java class is always interpreted with the same copy of $\mathbb{N}$. Each element of $\mathcal{S}_{\mathrm{DL}}$ describes a snapshot that might occur during execution of a program (without information about the program itself and the instruction pointer), namely the contents of the heap (values of attributes) and the values of variables.

If $S, S' \in \mathcal{S}_{\mathrm{DL}}$ are two such states, then we write $S \xrightarrow{p} S'$ to denote that the execution of the Java program p—when starting with the memory contents described by S—terminates in state S'. The evaluation of formulas can directly be defined based on this notion of state transition:

[2] JavaDL strictly distinguishes formulas and terms of sort *boolean*; the latter ones are used to gain a consistent mapping of Java expressions to first-order terms.

Definition 1 (JavaDL Semantics): Suppose that $S \in \mathcal{S}_{\mathrm{DL}}$ is a program state. The value $\mathrm{val}_S(\varphi)$ of a JavaDL formula φ as an element of $\{\mathrm{ff}, \mathrm{tt}\}$ is inductively defined by

$$\mathrm{val}_S(\langle p \rangle \varphi) = \mathrm{tt} \quad \text{iff there is } S' \in \mathcal{S}_{\mathrm{DL}}\colon \quad S \xrightarrow{p} S',\ \mathrm{val}_{S'}(\varphi) = \mathrm{tt}$$

and as is common for all first-order connectives (see for instance [Fit96]).

A central observation is that adjacent worlds $S, S' \in \mathcal{S}_{\mathrm{DL}}$, i.e. worlds with $S \xrightarrow{p} S'$ for an arbitrary program p, are *finite variants* [HKT00] of each other. This means that S, S' agree on the interpretations of symbols for all but finitely many points, which follows from the fact that a program can in finitely many execution steps only change the values of finitely many data locations.

3.3 Calculi for the Validity Problem in JavaDL

JavaDL contains arithmetic—as a necessary premise for modelling Java—which directly entails that there are no sound and complete (and finitary) calculi for validity. The usual way of coping with this fact is to assume the existence of an oracle handling the arithmetic fragment of a dynamic logic. For JavaDL, this oracle would be required to semi-decide the validity problem for JavaDL formulas that do not contain any modal operators. Adding modal operators with Java programs does not increase the expressiveness of the logic, and it is possible to construct *relatively complete* calculi for JavaDL validity that translate formulas containing modal operators to formulas without. See [HKT00, Pla04] for a treatment of this topic.

4 Counterexamples

According to the definition of validity in the previous sections, a JavaDL formula φ is *invalid* if there is a particular state $S \in \mathcal{S}_{\mathrm{DL}}$ such that $\mathrm{val}_S(\varphi) = \mathrm{ff}$, or equivalently if $\neg\varphi$ is satisfiable. For a formula that is potentially invalid, it is consequently interesting (i) to investigate whether an invalidating structure exists at all, and, more specifically, (ii) to determine such a structure or a set of such structures as counterexample for φ.

Definition 2 (Counterexample): A *counterexample* of a JavaDL formula φ is an algebra $S \in \mathcal{S}_{\mathrm{DL}}$ over the vocabulary built in Sect. 3.1 such that $\mathrm{val}_S(\varphi) = \mathrm{ff}$.

4.1 Derivation of Counterexamples

Because the domains of the algebras that are potential counterexamples for a formula are fixed, an invalidating one is described uniquely by the interpretations of existing function symbols, i.e. by the values of defined attributes and of variables. In the presence of higher-order quantifiers and for a vocabulary containing the functions $f_1, \ldots, f_n$, a formula φ is consequently invalid if and only if the formula

$$\psi \;:=\; \exists f_1 \ldots \exists f_n . \neg\varphi \tag{1}$$

is valid (or satisfiable, which is equivalent).[3] The values of $f_1, \ldots, f_n$ range over the corresponding possible operations on the algebra domain, and a counterexample for φ is described by matching choices for $f_1, \ldots, f_n$.

[3]The symbols $f_1, \ldots, f_n$ are shamelessly reused and bound instead of introducing higher-order variables for this purpose.

ψ is in general second-order, however, which means that proving its validity can be more difficult than the validity problem for JavaDL: a relatively complete calculus would require a more powerful (or more mysterious) oracle for higher-order formulas. While this could be viable for a large number of practical verification problems, it would not be possible to directly use implementations of JavaDL validity calculi to this end. Thus, the next section proposes a setting in which higher-order quantification can be replaced with first-order quantification over algebraic data types.

4.2 Restriction to a Fragment of JavaDL

In order to reduce the complexity of showing formulas φ invalid, we first alter the concept of a JavaDL structure of Sect. 3.2 slightly. This is possible because the original definition is very general and also admits program states that cannot occur in reality: during the actual execution of a program there will never be infinitely many instantiated objects, and hence only finitely many points of the attribute vectors $a : C \to T$ introduced in Sect. 3.1.2 are really used to store data. A program p that is started in an initial state $S_0 \in \mathcal{S}_{\mathrm{DL}}$ only reaches states $S \in \mathcal{S}_{\mathrm{DL}}$ that are finite variants of S_0.

Instead of taking all (usually uncountably many) algebras $\mathcal{S}_{\mathrm{DL}}$ into account, consequently we only examine the subset $\mathcal{S}_{\mathrm{DL}}^{\mathrm{Fin}} \subseteq \mathcal{S}_{\mathrm{DL}}$ of finite variants of one particular and fixed algebra $S_0 \in \mathcal{S}_{\mathrm{DL}}$. If one is interested in investigating the behaviour of programs in a realistic context, then a suitable choice for S_0 is a structure in which the elements of all attribute vectors are set to some standard value, and in which in particular the value of the implicit attributes `<created>` is always `FALSE` (no objects are instantiated). This is equivalent to requiring that in $\mathcal{S}_{\mathrm{DL}}^{\mathrm{Fin}}$ instance attributes of classes are always assigned some standard value for all but finitely many objects.

The modification of the semantics can be observed by formulas. Based on an initial state S_0 in which no objects are instantiated, and assuming that a Java class C is declared, for instance the formula

$$\neg\, \forall\, o : C.\ \texttt{<created>}(o) \doteq \texttt{TRUE}$$

is valid in $\mathcal{S}_{\mathrm{DL}}^{\mathrm{Fin}}$ but not in $\mathcal{S}_{\mathrm{DL}}$. Formulas like the one shown are not very meaningful when making statements about program states, however, and are consequently regarded as ill-formed and not taken into account: besides other restrictions, proper formulas only talk about existing objects. Sufficient syntactic criteria for well-formedness (for a similar purpose) are for instance discussed in [Pla04].

The finite variant restriction entails that the range of the quantification $\exists f_1, \ldots, \exists f_n.$ in Eq. (1) can be diminished: now it is only necessary to quantify over finitely (but unboundedly) many points of $f_1, \ldots, f_n$, because only those interpretations of the symbols have to be covered that can actually turn up in algebras of $\mathcal{S}_{\mathrm{DL}}^{\mathrm{Fin}}$. In the presence of arithmetic, such a quantification can be performed using first-order quantifiers, for instance by quantification over lists.

The oracles that are employed in Sect. 3.3 for examining the validity of JavaDL formulas are able to handle arithmetic, and can thus be expected also to cover algebraic datatypes (that in theory are not more difficult than arithmetic, as an encoding of datatypes into arithmetic is in the present situation always possible). In order to obtain a relatively complete calculus for the invalidity of JavaDL formulas φ in the described

fragment, one can therefore (i) derive the negated and quantified formula of Eq. (1), (ii) replace the higher-order quantification with algebraic datatypes (which is discussed in the next section), and (iii) use a relatively complete calculus to prove the formula valid.

4.3 From Higher-Order Quantifiers to Quantified Updates

In order to assign the functions $f_1, \ldots, f_n$ values that are determined by algebraic data structures, it is necessary to introduce a further operator, the *update operator* [Bec01, Pla04]. Updates are in JavaDL usually employed to memorise the effect of assignments in programs and enable an efficient treatment of the sequentiality inherent in imperative programming languages.

In the simplest case, terms or formulas can be preceded with expressions of the shape $\{f(s_1, \ldots, s_k) := t\}$ that cause one location of a function f to be updated to the value of t. A valid formula is for instance

$$\{f(1) := 1\}\big(1 \doteq f(f(1))\big)$$

When dealing with the quantifiers $\exists f_1 \ldots \exists f_n.$, the number of cells that need to be assigned is unbounded, which necessitates the usage of a more general kind of updates [Rüm05]. *Quantified updates* allow expressions $\{\texttt{for}\ i \bullet f(s_1, \ldots, s_k) := t\}$, where the variable i may occur in the terms $s_1, \ldots, s_k, t$, and can alter the values of unboundedly or infinitely many locations. A valid formula involving a quantified update is

$$\{\texttt{for}\ i : \mathit{int} \bullet f(i) := 2 \cdot i + 1\}\big(f(f(3)) \doteq 15\big)$$

Being equipped with updates, the particular quantifiers $\exists f_k.$ can be replaced with first-order quantification as follows: the only interesting case are symbols $f_k = a$ modelling instance attributes of classes, i.e. symbols $a : C \to T$. As a first solution, we make use of the algebraic datatype of lists, which can be finitely axiomatised by employing arithmetic.[4] We assume that the datatype is provided through a sort *ListOfT* representing lists of T-values, and through an access operator $l \downarrow i$ that returns the ith element of a list l. To circumvent some technicalities, the operator is required to be total and to return the standard value of attribute a (as in Sect. 4.2) when trying to access non-existing components of l (i is too large).

A formula $\exists a.\varphi$ can then be rewritten to

$$\psi \quad := \quad \exists\, l : \mathit{ListOfT}.\ \{\texttt{for}\ i : \mathit{nat} \bullet a(\mathit{obj}_C(i)) := l \downarrow i\}\ \varphi, \tag{2}$$

i.e. the attribute vector a is updated with the values of list l, and with its standard value in all places that are not determined by l. To see that the rewritten quantification has the right effect, one can recall that the evaluation of formulas takes place in algebras $S \in \mathcal{S}_{\mathrm{DL}}^{\mathrm{Fin}}$, i.e. the symbol a is always interpreted with an attribute vector a^S that contains some standard value in almost all places. The quantified update $\{\texttt{for}\ i : \mathit{nat} \bullet a(\mathit{obj}_C(i)) := l \downarrow i\}$ assigns a such a vector,[5] and all possible vectors can on the other hand be reached by choosing an appropriate list l.

[4] In theory, it would equivalently be possible to use an encoding of lists into natural numbers.

[5] The update can also be regarded as binding the symbol a within its scope: renaming a to some other symbol (that does not already turn up in φ) does not change the meaning of the formula. This explains why a seems to occur free in ψ but not in $\exists a.\varphi$.

4.4 Searching For Counterexamples Using Free Variables

After having produced a formula like Eq. (2), one is particularly interested in finding satisfying values of the variable l: the lists that l is ranging over represent (parts of) counterexample candidates for the original formula, namely the values of attribute a for finitely many objects. The derivation of the lists can practically be performed by using a tableaux-like calculus with existential free variables [Fit96]. In this setting, l is in the beginning of a proof replaced with a place holder, a free variable L. From a closed proof one can then read off the desired list by looking at the term that was substituted for L. If the substitution that closed the proof is not ground, then it can even be regarded as a description of a class of counterexamples.

4.5 Different Algebraic Datatypes

Lists are the most obvious algebraic datatype for representing the values of instance attributes, but by far not the only possible one. A different choice would be a datatype for partial functions with finite graph, which would suite the task of specifying the values of attributes for finitely many objects more directly. It can be expected that one needs to make a trade-off between (i) types that enable a good representation of natural classes of counterexamples, and (ii) types that make efficient search for counterexamples possible. Lists have some restrictions regarding the first item, because they make a representation of counterexamples modulo permutation of objects difficult.

5 Example

As test-bed for the described approach the KeY prover [ABB+05] was used, which provides an almost complete implementation of a JavaDL calculus (and a complete implementation for the Java variant JavaCard for smart card software). KeY contains a concept of free existential variables (called metavariables) that work well together with the quantification over algebraic datatypes.

The following paragraphs show an example of disproving a JavaDL formula given in the syntax that is used by the KeY system. Context of the example is a class `C`:

```
public class C { public int attr; }
```

and two variables `x`, `y` of type `C`. In this environment we specify that a piece of Java code swaps the values of the integer attribute of two objects by doing a bit of arithmetic:

```
!x=null & !y=null & x.<created>=TRUE & y.<created>=TRUE ->
ex xPre:int. ex yPre:int. ( x.attr = xPre & y.attr = yPre &
  <{ y.attr += x.attr;
     x.attr  = y.attr - x.attr;
     y.attr -= x.attr; }> ( x.attr = yPre & y.attr = xPre ) )
```

The task of finding the bug in the program (or in the specification) is left to the reader for the time being.

In order to show that this formula is invalid, it is negated and quantifiers are added for all relevant symbols, i.e. for the variables `x`, `y` and the attribute `attr`. The two local

variables are treated by two quantifiers (resp.) for covering also the possible value `null`.[6] Accesses to list elements are written `al(Cattr,i)`.

```
ex Cattr:ListOfInt.        ex xId:nat.       ex yId:nat.
ex Ccreated:ListOfBoolean. ex xNull:boolean. ex yNull:boolean.
{ for (i:nat) objC(i).attr := al(Cattr, i),
  for (i:nat) objC(i).<created> := al(Ccreated, i),
  x := if (xNull=TRUE) (null) (objC(xId)),
  y := if (yNull=TRUE) (null) (objC(yId)) }
!( !x=null & !y=null & x.<created>=TRUE & y.<created>=TRUE ->
ex xPre:int. ex yPre:int. ( x.attr = xPre & y.attr = yPre &
  <{ y.attr += x.attr;
     x.attr  = y.attr - x.attr;
     y.attr -= x.attr; }> ( x.attr = yPre & y.attr = xPre )))
```

This formula can be proven valid using KeY in a mostly automated fashion (which means that the considered program violates its specification). In order to illustrate what is happening we show the major steps of a proof. The calculus used is a Gentzen-style sequent calculus for JavaDL [Bec01].

First, the quantifiers can be removed by introducing metavariables (place holders `XID`, `YID`, `XNULL`, `YNULL`, `CATTR`, `CCREATED` for the counterexample) and skolem symbols (denoted with `@xPre`, `@yPre`), and after a number of basic simplifications the proof arrives at two goals. The first goal ensures that the counterexample that is searched for is consistent with the precondition of the original formula. The second goal states that the program does not terminate properly, i.e. that it does not terminate or that the postcondition is violated after its termination.

```
==> XNULL = FALSE             & YNULL = FALSE &
    al(CCREATED, XID) = TRUE & al(CCREATED, YID) = TRUE
=============================================================
 al(CATTR, XID) = @xPre,   al(CATTR, YID) = @yPre,
 XNULL = FALSE,            YNULL = FALSE,
 al(CCREATED, XID) = TRUE, al(CCREATED, YID) = TRUE
==> ! { for (i:nat) objC(i).attr := al(CATTR, i),
        for (i:nat) objC(i).<created> := al(CCREATED, i),
        x := if (XNULL=TRUE) (null) (objC(XID)),
        y := if (YNULL=TRUE) (null) (objC(YID)) }
     <{ y.attr += x.attr;
        x.attr  = y.attr - x.attr;
        y.attr -= x.attr; }> (x.attr = @yPre & y.attr = @xPre)
```

In the second sequent it is now possible to work off the Java program by symbolic execution. This leads to two new goals in the end, because a case distinction between `XID=YID` and `!XID=YID` is made: the expressions `x.attr`, `y.attr` in the program can either address the same or different locations.

[6] This could be done in other, perhaps more elegant ways as well, but until now it is not clear which possibility is the most efficient one in practice.

```
==> XNULL = FALSE             & YNULL = FALSE &
    al(CCREATED, XID) = TRUE & al(CCREATED, YID) = TRUE
============================================================
 @xPre=0, @yPre=0, XID=YID, ...
==> ! al(CATTR, XID) = 0
============================================================
 @xPre=al(CATTR, XID), @yPre=al(CATTR, YID), !XID=YID, ...
==> ! al(CATTR, YID) + al(CATTR, XID) - al(CATTR, YID)
        = al(CATTR, XID)
```

At this point a counterexample can be constructed by choosing suitable instantiations of the free variables (KeY is able to find these instantiations automatically). The last goal can be closed by the substitution [`YID/XID`] (`XID` is substituted for `YID`) because of the negated equation of the antecedent. To handle the second goal, the equation `al(CATTR,XID)=0` can be made wrong by (for instance) choosing `CATTR` as the list `[1]` and `XID` as `0`. Finally, the first goal can be handled by substituting `FALSE` for `XNULL`, `YNULL` and `[TRUE]` for `CCREATED`. Altogether, the proof is closed by the substitution

[`XID/0`, `YID/0`, `CATTR/[1]`, `CCREATED/[TRUE]`, `XNULL/FALSE`, `YNULL/FALSE`]

When having a closer look at the program, one can see that the program violates its specification whenever started in a state in which (i) the variables `x` and `y` point to the same object, and (ii) the value of `x.attr` (which is the same as `y.attr`) is not `0`. The counterexample described by the substitution is one particular instance of this situation.

6 Conclusions and Future Work

A method has been presented that uses calculi for the validity problem for Java Dynamic Logic to disprove formulas, and that is also able to construct counterexamples or whole classes of counterexamples. Within a fragment of JavaDL the method provides relatively complete calculi for satisfiability. Only little implementation work has to be invested to make a theorem prover for JavaDL (in our case KeY) search for counterexamples, and the procedure immediately supports full Java if this is the case for the underlying prover.

Our method is particularly suited for showing that programs are partially incorrect, i.e. that there are inputs for which a program terminates but produces wrong results, because in this setting loops can be treated without inductive arguments: for a particular counterexample one can only have a bounded number of loop iterations. Treating total incorrectness is most likely a more difficult problem (at least in practice) which we plan to investigate in the future.

Though the experiments with finding counterexamples by employing the KeY prover and the presented method are promising so far, it remains to be examined how well the method performs for more complex programs. We expect that this will necessitate a number of optimisations of KeY specifically for counterexample search. In the end it would be interesting to compare the performance of our approach with existing techniques for generating test cases. While the approach described here has a very high

precision and the ability to construct whole classes of counterexamples, the complexity caused by using a theorem prover and symbolic execution to this end is significant. Techniques like random testing can be expected to find counterexamples with a much lower effort in many cases, but are bound to fail in situations in which counterexamples are only sparsely distributed over the space of all program inputs and hard to find. In practice it should be beneficial to combine these different paradigms for debugging software, which is one further aspect we consider as future work.

7 Related Work

Model finders like Paradox [CS03] are able to derive finite models of conjectures (provided that there are any). Because JavaDL involves arithmetic, model finders are not directly applicable for the derivation of counterexamples as described in this abstract, but could be used in combination with abstraction techniques.

In [Ahr02], models of conjectures are derived by searching for appropriate valuations of loose functions over freely generated algebras. The method is complete but in general not sound, i.e. it is possible that false models are found. Still it might be possible to use the approach in combination with the ideas given in this abstract to extract counterexamples.

Software testing is concerned with finding input data for a given program that reveals erroneous behaviour. The generation of these input records can be guided in different ways, for instance by creating random data [CH00], by exhaustively covering non-isomorphic classes of input data [BKM02, MK01] or by trying to optimise the execution path coverage in some way. For the latter approach, both actual and symbolic execution of programs are employed [GMS98, BCH$^+$04], and also model-checking techniques are used. Related to our approach is [VPK04], which—besides comparing different approaches for using a model-checker (Java PathFinder [VHBP00]) for creating test input sets with maximal code coverage—focuses on a combination of model-checking and symbolic execution to treat branch predicates and preconditions. In contrast to our method, conditions that are symbolically derived are only used to create test input, and are not verified against a specification.

ESC/Java2 [SoSGox] uses a theorem prover to detect violations of (implicit or explicit) assertions that are added to Java code. The prover Simplify is able to derive counterexamples from failed prove attempts, which are subsequently used to create warnings about possible erroneous behaviour of a program for certain concrete situations.

In [RST01], failed proof attempts of the theorem prover KIV [BRS$^+$00] are used to derive counterexamples for theorems concerning algebraic datatypes and dynamic logic. The approach reuses an existing calculus for the validity problem and is not restricted to a fragment of the concerned logic, but apparently there are no completeness results.

Acknowledgements

I would like to thank Wolfgang Ahrendt for many comments and discussions that led to this paper, and Wojciech Mostowski and Tobias Gedell for numerous complains and discussions about unclear paragraphs. Thanks are also due to the anonymous referees for helpful comments.

References

[ABB+05] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.

[Ahr02] Wolfgang Ahrendt. Deductive search for errors in free data type specifications using model generation. In Andrei Voronkov, editor, *Automated Deduction – CADE-18, 18th International Conference on Automated Deduction, Copenhagen, Denmark*, volume 2392 of *LNCS*. Springer-Verlag, 2002.

[BCH+04] Dirk Beyer, Adam J. Chlipala, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Generating tests from counterexamples. In *Proceedings of the 26th International Conference on Software Engineering (ICSE'04, Edinburgh)*, pages 326–335. IEEE Computer Society Press, 2004.

[Bec01] Bernhard Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, volume 2041 of *LNCS*, pages 6–24. Springer, 2001.

[BKM02] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: automated testing based on java predicates. *SIGSOFT Softw. Eng. Notes*, 27(4):123–133, 2002.

[BRS+00] M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal system development with KIV. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, volume 1783 of *LNCS*. Springer, 2000.

[CH00] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *ACM SIGPLAN Notices*, 35(9):268–279, 2000.

[CS03] Koen Claessen and Niklas Sörensson. New techniques that improve mace-style finite model finding. In Peter Baumgartner and Chris Fermueller, editors, *Model Computation - Principles, Algorithms, Applications*, Miami, Florida, July 2003. CADE-19 Workshop.

[Fit96] Melvin C. Fitting. *First-Order Logic and Automated Theorem Proving*. Springer-Verlag, New York, second edition, 1996.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison Wesley, 2nd edition, 2000.

[GMS98] Neelam Gupta, Aditya P. Mathur, and Mary Lou Soffa. Automated test data generation using an iterative relaxation method. In *SIGSOFT '98/FSE-6: Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 231–244. ACM Press, 1998.

[HKT00] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. MIT Press, 2000.

[MK01] Darko Marinov and Sarfraz Khurshid. Testera: A novel framework for automated testing of java programs. In *ASE '01: Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, page 22. IEEE Computer Society, 2001.

[Pla04] André Platzer. An object-oriented dynamic logic with updates. Master's thesis, University of Karlsruhe, Department of Computer Science. Institute for Logic, Complexity and Deduction Systems, September 2004. `http://www.functologic.com/logic/Diplomath.pdf`.

[RST01] Wolfgang Reif, Gerhard Schellhorn, and Andreas Thums. Flaw detection in formal specifications. In *IJCAR '01: Proceedings of the First International Joint Conference on Automated Reasoning*, pages 642–657. Springer-Verlag, 2001.

[Rüm05] Philipp Rümmer. A language for sequential, parallel and quantified updates of first-order structures. To appear, 2005. To appear.

[Sch02] Steffen Schlager. Behandlung von Integer Arithmetik bei der Verifikation von Java-Programmen. Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe, May 2002.

[SoSGox] University of Nijmegen Security of Systems Group. ESC/Java2 (Extended Static Checking for Java Version 2) project, . `http://www.sos.cs.ru.nl/research/escjava/index.html`.

[VHBP00] Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Model checking programs. In *ASE '00: Proceedings of the The Fifteenth IEEE International Conference on Automated Software Engineering (ASE'00)*, page 3. IEEE Computer Society, 2000.

[VPK04] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. Test input generation with java pathfinder. *SIGSOFT Softw. Eng. Notes*, 29(4):97–107, 2004.

Automatic theorem proving for program verification engines

Byron Cook
Microsoft Research
Cambridge

Invited talk, joint with the workshop on *Empirically Successful Classical Automated Reasoning (ESCAR)*

Abstract. Microsoft has made an extensive investment in the area of software model checking. Tools such as Slam, Zing, KIS, Terminator, and ESP are now being used both inside and outside of the company. In this talk I'll describe some of the underlying automatic theorem proving frastructure used by these verification tools. I will also describe some recent advances and findings in this area.

Bio: Dr. Byron Cook is a researcher at Microsoft's research laboratory in Cambridge, England. His research interest include automatic theorem proving, model checking and programming language theory.