CAV 2008

20th International Conference on
Computer-Aided Verification

July 2008
Princeton, New Jersey

SMT 2008:
6th International Workshop on Satisfiability Modulo Theories

July 7-8, 2008
Proceedings

Editors:
Clark Barrett, Leonardo de Moura

Workshop Chairs

Clark Barrett
New York University

Leonardo de Moura
Microsoft Research

Program Committee

Nikolaj Bjørner
Microsoft Research

Alessandro Cimatti
ITC-Irst, Trento

Bruno Dutertre
SRI International

Sava Krstic
Intel Corporation

Robert Nieuwenhuis
Techn. Univ. of Catalonia

Albert Oliveras
Techn. Univ. of Catalonia

Silvio Ranise
LORIA, Nancy

Roberto Sebastiani
Università di Trento

Ofer Strichman
Technion

Aaron Stump
University of Iowa

Cesare Tinelli
University of Iowa

Preface

This volume contains the proceedings of SMT 2008, the 6th International Workshop on Satisfiability Modulo Theories, held in Princeton, New Jersey on July 7-8, 2008. The workshop was affiliated with the 20th International Conference on Computer-Aided Verification (CAV 2008).

The primary goal of the workshop was to bring together both researchers and users of SMT technology and provide them with a forum for presenting and discussing theoretical ideas, implementation and evaluation techniques, and applications. The workshop also served as a forum for further discussion of the Satisfiability Modulo Theories Library (SMT-LIB) initiative that aims at establishing a common standard for the specification of benchmarks and background theories for SMT. Finally, continuing the pattern of SMT 2007, the workshop included sessions dedicated to the SMT competition (SMT-COMP 2008).

There were three categories of technical papers: *extended abstracts* containing preliminary reports of work in progress; *original papers* containing original research; and *presentation-only papers* describing work recently published or submitted elsewhere but of sufficient interest to the SMT community to warrant a presentation at the workshop. This volume includes the full text of the extended abstracts and original papers, but only the abstracts of presentation-only papers.

The final program included:

- invited talks by Jean-Christophe Filliâtre (Laboratoire de Recherche en Informatique - Université Paris Sud), Aaron Bradley (University of Colorado at Boulder), and Nikolai Tillmann (Microsoft Research)
- 13 technical paper presentations, including 5 original papers, 7 presentation-only papers, and 1 extended abstract
- two panels: one on SMT-LIB and one on SMT-COMP
- a session in which participants in SMT-COMP briefly presented their tools.

Additional details for the workshop including the program are available at the web site `http://research.microsoft.com/conferences/SMT08/`.

We gratefully acknowledge the financial support of Microsoft Research and Intel Corporation.

Princeton, July 2008 Clark Barrett

Leonardo de Moura

Table of Contents

Using SMT Solvers for Deductive Verification of C and Java Programs *(invited talk)* 1
Jean-Christophe Filliâtre

Implementing Polymorphism in SMT solvers *(original paper)* 2
Francois Bobot, Sylvain Conchon, Evelyne Contejean, Stéphane Lescuyer

Cover Algorithms and Their Combination *(presentation-only paper)* ... 12
Sumit Gulwani, Madanlal Musuvathi

SAT Modulo the Theory of Linear Arithmetic: Exact, Inexact and Commercial Solvers *(presentation-only paper)* 13
Germain Faure, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez Carbonell

An Algebraic Approach for the Unsatisfiability of Nonlinear Constraints *(presentation-only paper)* 14
Ashish Tiwari

Using an SMT Solver and Craig Interpolation to Detect and Remove Redundant Linear Constraints in Representations of Non-Convex Polyhedra *(original paper)* 15
Christoph Scholl, Stefan Disch, Florian Pigorsch

Efficient Interpolant Generation in Satisfiability Modulo Theories *(presentation-only paper)* 31
Alessandro Cimatti, Alberto Griggio, Roberto Sebastiani

Rocket-Fast Proof Checking for SMT Solvers *(presentation-only paper)* 32
Michał Moskal

Proof Translation and SMT-LIB Benchmark Certification: A Preliminary Report *(extended abstract)* 33
Yeting Ge, Clark Barrett

Towards an SMT Proof Format *(original paper)* 44
Aaron Stump, Duckki Oe

Reasoning about Arrays *(invited talk)* 57
Aaron Bradley

Lemmas on Demand for the Extensional Theory of Arrays *(original paper)* 58
Robert Brummayer, Armin Biere

Towards SMT Model Checking of Array-based Systems *(presentation-only paper)* 68
Silvio Ghilardi, Enrica Nicolini, Silvio Ranise, Daniele Zucchelli

Back to the future: Revisiting Precise Program Verification using SMT solvers *(presentation-only paper)* 69
Shuvendu K. Lahiri, Shaz Qadeer

SMELS: Satisfiability Modulo Equality with Lazy Superposition *(presentation-only paper)* 70
Christopher Lynch, Duc-Khanh Tran

Deciding Array Formulas with Fruagal Axiom Instantiation *(original paper)* 71
Amit Goel, Sava Krstić, Alexander Fuchs

Pex - White Box Test Generation for .NET *(invited talk)* 82
Nikolai Tillmann

Using SMT solvers for deductive verification of C and Java programs

Jean-Christophe Filliâtre

Laboratoire de Recherche en Informatique (Université Paris Sud), France
Jean-Christophe.Filliatre@lri.fr

Abstract. This talk details the use of SMT solvers for the verification of C and Java programs in Why platform http://why.lri.fr/. In particular, we will explain the choice of a polymorphic first-order logic as a common input language for the various SMT solvers, and how we encode this logic when the SMT solver does not handle it natively. We will also review the relevant theories for program verification, and how axiomatic models can be used as workarounds when built-in theories are missing in SMT solvers. Finally, we also explain how we use SMT solvers from the interactive proof assistant Coq, to handle verification conditions that could not be discharged automatically.

Implementing Polymorphism in SMT solvers*

François Bobot Sylvain Conchon Evelyne Contejean
Stéphane Lescuyer

LRI, Univ. Paris-Sud, CNRS, Orsay F-91405
& INRIA Futurs, ProVal, Orsay, F-91893
FRANCE

Abstract

Based on our experience with the development of Alt-Ergo, we show a small number of modifications needed to bring parametric polymorphism to our SMT solver. The first one occurs in the typing module where unification is now necessary for solving polymorphic constraints over types. The second one consists in extending triggers' definition in order to deal with both term and type variables. Last, the matching module must be modified to account for the instantiation of type variables. We hope that this experience is convincing enough to raise interest for polymorphism in the SMT community.

1 Introduction

The SMT-LIB [12] provides **ArraysEx**, a generic theory of arrays with extensionality, which introduces three sorts for *indices*, *elements* and *arrays*. Unfortunately, the sorts *indices* and *elements* are not parameters but constant types. Therefore, typing constraints prevent one from using a decision procedure of **ArraysEx** for arrays of integers, even though such a decision procedure would not depend on the sorts of indices and elements. Three other variants are thus provided by SMT-LIB: arrays containing integers (**Int_ArraysEx**), bit vectors (**BitVector_ArraysEx**) and arrays of reals (**Int_Int_Real_Array_ArraysEx**).

This replication issue also occurs with user-defined specifications. For instance, in the context of program verification, the formalization of memory models consists in a set of definitions and axioms which are actually independent of the type of memory cell contents [4, 9].

Polymorphic types [11] are an elegant solution to this problem: they allow a single set of definitions and axioms to be used with different types of data. Aside from the obvious gain of conciseness offered by such factoring of specifications, polymorphism also offers additional expressiveness, for instance the use of type variables as *phantom types* to ensure separation in memory models [6, 13].

In this paper, we show that only a small number of modifications are needed to bring polymorphic types to an SMT solver. In the first part, we detail the example

*Work partially supported by A3PAT project of the French ANR (ANR-05-BLAN-0146-01).

of array theory so as to describe our polymorphic logic and its subtleties. In thesecond part, we describe in turn how the typing and matching mechanisms ought to be modified in order to deal with polymorphic theories. Finally, we argue why adding polymorphism to a solver is a better solution than other existing workarounds.

2 Polymorphic First-Order Logic

A Case Study: Theory of Arrays. In order to show how polymorphic types can circumvent the duplication of types and axioms, we transform the **ArrayEx** theory of the SMT-LIB given in Alt-Ergo-syntax in Figure 1 into a theory of arrays using polymorphic types (Figure 2). This theory will illustrate how a single set of axioms and definitions can be used to prove goals over several kinds of arrays (containing integers, bitvectors, other arrays, *et caetera*).

```
type array
type index
type element

logic select : array, index → element
logic store : array, index, element → array

axiom a1:
  forall a:array. forall i:index. forall e:element.
   select((store(a,i,e),i) = e

axiom a2:
  forall a:array. forall i,j:index. forall e:element.
   i<>j → select(store(a,i,e),j) = select(a,j)

axiom a3:
 forall a,b:array. (forall i:index. select(a,i)=select(b,i)) → a=b
```

Figure 1: The SMT-LIB theory of arrays in Alt-Ergo syntax

The first three lines in Figure 1 define the abstract sorts `array`, `index` and `element`. The next two lines define the signature of the usual functions `select` and `store` over arrays: given an `array` *a* and an `index` *i*, `select`(*a*,*i*) returns the *i*-th `element` of *a*, whereas `store`(*a*,*i*,*e*) returns the array *a* updated with *e* at index *i*.

The abstract nature of `index` and `element` prevents one to use this theory over actual arrays, *e.g.* of integers, since integers are of sort `int` and not `element`. It is desirable that `element` could be replaced by `int` (or any other sort), and similarly for indices. A well-known answer to this requirement is parametric polymorphism [11].

Indeed, it can be noticed that the axioms `a1`, `a2` and `a3` do not depend on the particular nature of indices and elements. As long as the latter can be compared

with respect to equality, they can be seen as black boxes. Informally, this means that the only sort which is specified by the theory is `array`, thus one would rather define a family of arrays parameterized by sorts for indices and elements. In Alt-Ergo syntax, this family is denoted by:

```
type ('i,'e) array
```

where `'i` and `'e` are type variables representing respectively indices and elements.

The signatures of functions and axioms have to be modified accordingly, by replacing every occurrence of `array` (resp. `index`, resp. `element`) by `('i,'e) array` (resp. `'i`, resp. `'e`). The resulting polymorphic theory is shown in Figure 2.

```
type ('i,'e) array

logic select : ('i,'e) array, 'i → 'e
logic store  : ('i,'e) array, 'i, 'e → ('i,'e) array

axiom a1 :
  forall a:('i,'e) array. forall i:'i. forall e:'e.
  select((store(a,i,e),i) = e

axiom a2 :
  forall a:('i,'e) array. forall i,j:'i. forall e:'e.
  i<>j → select(store(a,i,e),j) = select(a,j)

axiom a3 :
 forall a,b:('i,'e) array. (forall i:'i. select(a,i)=select(b,i)) → a=b
```

Figure 2: The polymorphic theory of arrays

This parametric theory of arrays can be instantiated, for example to retrieve SMT-LIB's arrays:

ArraysEx	`(index, element) array`
Int_ArraysEx	`(int, int) array`
BitVector_ArraysEx	`(bitvector, bitvector) array`
Int_Int_Real_Array_ArraysEx	`(int, (int, real) array) array`

Moreover, several distinct theory instances can be used conjointly, as illustrated in the following formula:

```
goal g1:
 forall i,j:int.
 forall m:(int,(int,int) Array) Array. forall r:(int,int) Array.
 r = select(m,i) → store(m,i,store(r,j,select(r,j))) = m
```

This goal involves two different instances of the theory of arrays, and mixes function symbols from both theories. Indeed, in the conclusion of this goal, the outer occurrence of the `store` function belongs to the theory of arrays of integer

arrays (integer matrices), whereas the inner occurrence of `store` belongs to the theory of integer arrays. It is worth noting that there is no syntactic distinction between these two instances of the same parametric function symbol. We explain in Section 3 how the typing system finds the correct instances for each function symbol.

Polymorphism and its Subtleties. Up to this point, we remained vague about the meaning of the free type variables in the definition of parametric symbols and polymorphic axioms. In order to highlight the subtleties introduced by polymorphism, we will now *explicitly* denote the *implicit* quantification of these type variables by using the symbol ∀ and type variables instantiation (in terms) by brackets. With these new conventions, the beginning of the arrays theory becomes:

```
type : ∀ 'i,'e. ('i,'e) array

logic select : ∀ 'i,'e. ('i,'e) array, 'i → 'e
logic store : ∀ 'i,'e. ('i,'e) array, 'i, 'e → ('i,'e) array

axiom a1 :
  ∀ 'i,'e. forall a:('i,'e) array. forall i:'i. forall e:'e.
  select['i,'e](store['i,'e](a,i,e),i) = e
```

The type `array` can be understood as a type family, *i. e.* a function yielding one array type for each pair of types `('i,'e)`. Similarly, `select` introduces a function family: for each possible `'i` and `'e`, it provides a function of type `('i,'e) array, 'i → 'e`.

The case of axiom `a1` is more informative: all the type variables in this axiom are universally quantified at the outer level of the definition. This outermost type quantification is general in Alt-Ergo and reflects our choice of prenex-polymorphism *a la* ML [10]. This choice will be discussed in Section 3. A very important consequence of this fact, and a major difference with the monomorphic multi-sorted first-order logic of the SMT-LIB, is that an axiom is different from an hypothesis in a goal. This can be seen in the following example:

```
type : ∀ 'c. 'c t
logic P : ∀ 'c. 'c t → prop.
axiom a : ∀ 'c. forall x: 'c t. P['c](x).
goal g2 : ∀ 'a,'b. (forall x: 'a t. P['a](x)) ∧ (forall x: 'b t. P['b](x))
```

This goal can be easily proved by instantiating axiom `a` once for each type variable `'a` and `'b`. Now consider putting this axiom as an hypothesis in the goal:

```
goal g3 : ∀ 'a,'b,'c.
  (forall x: 'c t. P['c](x)) →
  (forall x: 'a t. P['a](x)) ∧ (forall x: 'b t. P['b](x))
```

The goal is not valid anymore since it is only provable when `'a`, `'b` and `'c` are equal. This is a manifestation of the fact that in general the formulas $(\forall x, P \Rightarrow Q)$

and $(\forall x, P) \Rightarrow Q$ are not equivalent. Note, by the way, that even if the goal `g3` is polymorphic, a goal can always be considered monomorphic[1]: for every type variable `'a` universally quantified in the goal, it suffices to introduce a fresh ground type $t_{\texttt{a}}$ and to substitute `'a` by $t_{\texttt{a}}$. Proving this particular instantiation of the goal is equivalent to proving it for any instantiation, since no assumptions are made on the fresh types $t_{\texttt{a}}$.

The theory of polymorphic lists illustrates a new phenomenon which is not observable with arrays:

---

type `∀ 'a. 'a list`
logic `nil : ∀ 'a. 'a list`

In the above declaration, `nil` is a so-called *polymorphic constant*, that is a family of constants, one for each type. This implies that for each type `'a`, the type `'a list` is inhabited by $\texttt{nil}_{['\texttt{a}]}$, even if `'a` is not! A more pernicious declaration is

---

logic `any : ∀ 'a. 'a`

---

which makes *every* type inhabited. Polymorphic constants will have to be dealt with carefully during triggers' definition and matching (*cf.* Section 4).

We have now gained enough understanding about what parametric polymorphism means to be able to give a good intuition of the semantics of polymorphic first-order logic. Suppose we have a polymorphic theory $\mathbf{T}_{\text{PFOL}}$ and a goal $\mathbf{G}$; we wish to explain what it means for $\mathbf{T}_{\text{PFOL}}$ to *entail* $\mathbf{G}$, in symbols $\mathbf{T}_{\text{PFOL}} \models_{\text{PFOL}} \mathbf{G}$. As argued above, we can consider that $\mathbf{G}$ is monomorphic without restriction. Let $\mathcal{T}$ be the set of all ground types that can be built from the signature in $\mathbf{T}_{\text{PFOL}}$ along with an infinitely countable set of arbitrary fresh constant types[2]. Now, let us write $\mathbf{T}_{\text{FOL}}$ the monomorphic multi-sorted theory such that:

- the set of its types is $\mathcal{T}$;
- its function symbols are all monomorphic instances (with types in $\mathcal{T}$) of the function symbols defined in $\mathbf{T}_{\text{PFOL}}$;
- similarly, its axioms are all the possible monomorphic instances of the axioms in $\mathbf{T}_{\text{PFOL}}$.

Given such a theory, we have that $\mathbf{T}_{\text{PFOL}}$ entails $\mathbf{G}$ if and only if $\mathbf{T}_{\text{FOL}}$ entails $\mathbf{G}$ in monomorphic first-order logic, ie. $\mathbf{T}_{\text{PFOL}} \models_{\text{PFOL}} \mathbf{G} \Leftrightarrow \mathbf{T}_{\text{FOL}} \models_{\text{FOL}} \mathbf{G}$. Furthermore, since the proof of $\mathbf{G}$ in $\mathbf{T}_{\text{FOL}}$ is finite, only a finite number of monomorphic instances of the definitions in $\mathbf{T}_{\text{PFOL}}$ are necessary to establish a proof of $\mathbf{G}$. Altogether, this means that the task of solving a polymorphic problem amounts to finding the right monomorphic instances of the definitions in the problem and then solving the monomorphic problem we obtain in this manner. The task of generating and finding monomorphic instances is discussed in Sections 4.

[1] This means that pushing a polymorphic axiom into a goal makes it monomorphic.

[2] This technical requirement ensures that $\mathcal{T}$ be non-empty and that polymorphic goals can be monomorphized.

3 Type-checking

This section is devoted to the research of proper instances of polymorphic symbols occurring in a monomorphic formula (the case of polymorphic axioms will be treated in the next section). We chose the prenex-version of polymorphism since it is quite expressive and light to handle for a user, compared with polymorphism *a la system F*, which is more powerful, but requires types' annotations for *every* occurrence of polymorphic symbols to ensure a decidable type inference. A possible compromise would be polymorphism *a la* ML^F [8] where annotations are needed only when *defining* polymorphic symbols, but the whole machinery is much more complex than in the case of prenex-polymorphism.

We illustrate the well-known inference mechanism for prenex-polymorphism [11] on the goal `g1`, in particular on the subterm `select(m,i)`.

1. Since `select` is a polymorphic function, the first step is to build an instance with *fresh* type variables `x1` and `x2`, yielding $\texttt{select}_{[\texttt{x1},\texttt{x2}]}\texttt{(m,i)}$.

2. The constraints given by the signature of `select` are the following:

   $\texttt{select}_{[\texttt{x1},\ \texttt{x2}]}\texttt{(m,i)} : \texttt{x2} \qquad \texttt{m} : \texttt{(x1,x2) array} \qquad \texttt{i} : \texttt{x1}$

   Combined with the type annotations of `g1`:

   ```
   m : (int, (int,int) array) array      i : int
   ```

   we get that `x1 = int` and `x2 = (int, int) array` by unification [1].

Unification of the typing constraints enables finding (most general) instances such that all terms and formulas are well-typed. On the contrary, unification will fail if there is no way to find instances such that the formulas are well-typed. We can remark that the equality has type `∀'a. 'a, 'a -> prop`, so well-typedness ensures that every occurrence of an equality is homogeneous. In our example, the constraint that both terms `r` and `select(m,i)` have the same type is verified.

Introducing fresh type variables in the first step guarantees that different occurrences of the same polymorphic symbol can be instantiated independently, such as the two occurrences of `store` in goal `g1` or, more evidently, the occurrences of `x` in the following example:

---

type `'a t`

logic `x : ∀ 'a. 'a t`
logic `f : ∀ 'a. 'a t → 'a`
goal `g4 : ∀ 'a.` $\texttt{f(x}_{[\texttt{int}]}\texttt{) = 0 or f(x}_{[\texttt{real}]}\texttt{) = 4.5 or x}_{['\texttt{a}]}\texttt{=x}_{['\texttt{a}]}$ `or` $\texttt{f(x}_{[\texttt{int}]}\texttt{) = 3}$

The first and last `x` are the same instance `x[int]`, and thus represent the same constant. In `x=x`, the sole constraint on the type of `x` is that it be the same on both sides of the equality symbol. Therefore, both `x` are instantiated on the same fresh

(universally quantified) type variable `'a`, and after the monomorphization of the goal, this variable will become a new constant type, distinct from all other types.

After finding the right instances for all symbols, we must keep the types inferred in the abstract syntax tree (AST) of formulas, in order to be able to use this type information in the *matching* process, as explained in the next section.

4 Polymorphic Triggers and Matching

This section is illustrated by the theory of polymorphic lists, defined by

---

type ∀ 'a. 'a list
logic nil : ∀ 'a. 'a list
logic cons : ∀ 'a. 'a, 'a list → 'a list
logic length : ∀ 'a. 'a list → int

axiom l1 : ∀ 'a. $\texttt{length}_{[\texttt{'a}]}(\texttt{nil}_{[\texttt{'a}]}) = 0$
axiom l2 : ∀ 'a. forall x:'a. forall l:'a list.
$\texttt{length}_{[\texttt{'a}]}(\texttt{cons}_{[\texttt{'a}]}(\texttt{x,l})) = \texttt{length}_{[\texttt{'a}]}(\texttt{l}) + 1$

In order to prove the goal `g :` $\texttt{length}_{[\texttt{int}]}(\texttt{cons}_{[\texttt{int}]}(3,\texttt{nil}_{[\texttt{int}]}))=1$, a human being will first instantiate `l2` by $\mu_1 = \{\texttt{'a} \mapsto \texttt{int}\}$ and $\sigma_1 = \{\texttt{x} \mapsto 3, \texttt{l} \mapsto \texttt{nil}_{[\texttt{int}]}\}$. Then (s)he is left with $\texttt{length}_{[\texttt{int}]}(\texttt{nil}_{[\texttt{int}]})$ `+ 1 = 1`, which can be proved by using `l1` instantiated by $\mu_2 = \{\texttt{'a} \mapsto \texttt{int}\}$. An SMT solver needs to have a similar mechanism in order to "infer" the useful instances. This mechanism is based on so-called *triggers*, as presented in [3]. In this particular example, the solver "knows" a set of ground terms Γ, and terms are equipped with their type:

$\Gamma =$ $\texttt{length}_{[\texttt{int}]}(\texttt{cons}_{[\texttt{int}]}(3,\texttt{nil}_{[\texttt{int}]})) : \texttt{int}$, $\texttt{cons}_{[\texttt{int}]}(3,\texttt{nil}_{[\texttt{int}]}) : \texttt{int list}$,
$\texttt{1},\texttt{3} : \texttt{int}$, $\texttt{nil}_{[\texttt{int}]} : \texttt{int list}$

Given a lemma and its trigger, the matching module of Alt-Ergo searches Γ for an instance of the trigger (possibly modulo the equalities discovered so far), and applies the resulting substitutions for type and term variables to the lemma, yielding a ground *fact* which can feed the SAT-solver.

Assume that we have chosen as trigger for `l2` its subterm $\texttt{cons}_{[\texttt{'a}]}(\texttt{x,l})$, a possible instance is $\texttt{cons}_{[\texttt{int}]}(3,\texttt{nil}_{[\texttt{int}]})$, and the substitutions are μ_1 and σ_1. The matching module thus has to instantiate both type and term variables. When there are polymorphic constants, such as `any` or `nil`, although they are in a sense ground terms since they do not contain any *term* variables, they are not fully ground since their type may contain *type* variables. This explains why triggers can be terms without (term) variables. In particular, our axiom `l1` looks like it is a fact already, but it actually has to be instantiated before being fed to the SAT-solver. In our example, if the trigger of `l1` is $\texttt{length}_{[\texttt{'a}]}(\texttt{nil}_{[\texttt{'a}]})$, the solver can compute the instance μ_2.

We now turn to a more formalized definition of matching. Let us assume a set of *ground terms* Γ along with their types, for which we write $t : \tau \in \Gamma$ when

a ground term t of ground type τ belongs to Γ. We also consider an *equivalence relation* Δ on terms in Γ, representing the equalities discovered so far in the context of the solver. Given these sets, we first define a few notions, and go on to describing the matching algorithm itself:

- a *trigger* is a term where variables and constants are decorated with (possibly polymorphic) types: namely, we write $x^{\tau[\bar{\alpha}]}$ (resp. $c^{\tau[\bar{\alpha}]}$) when the variable x (resp. the constant c) is annotated with a type τ and $\bar{\alpha}$ are the free type variables in τ; given a trigger p, we write $\mathcal{V}(p)$ to denote the variables of p, and $\mathcal{TV}(p)$ the set of all type variables appearing in the variables of p;
- a *substitution* σ mapping variables $x_1,\ldots,x_n$ to terms $t_1,\ldots,t_n$ in Γ is denoted $\{x_1 \mapsto t_1;\ldots;x_n \mapsto t_n\}$; its *support* $Supp(\sigma)$ denotes the set of variables $x_1,\ldots,x_n$. Substitutions $\sigma_1,\ldots,\sigma_n$ are said to be Δ-*compatible* if the following holds:

$$\forall x\, i\, j, x \in Supp(\sigma_i) \cap Supp(\sigma_j) \Rightarrow \sigma_i(x) =_\Delta \sigma_j(x);$$

- a *type substitution* μ mapping some type variables $\bar{\alpha}$ to ground types $\bar{\tau}$ is denoted $\{\bar{\alpha} \mapsto \bar{\tau}\}$, and its support $Supp(\mu)$ is defined as above ; type substitutions $\mu_1,\ldots,\mu_n$ are said to be *compatible* if the following holds:

$$\forall \alpha\, i\, j, \alpha \in Supp(\mu_i) \cap Supp(\mu_j) \Rightarrow \mu_i(\alpha) = \mu_j(\alpha).$$

  Given n such compatible type substitutions, we write $\biguplus \mu_i$ the merging of these substitutions; similarly, we write $\biguplus \sigma_i$ the merging of Δ-compatible term substitutions.

The *matching function* $\mathcal{M}$ depends on Γ and Δ, ie. on the current ground environment of the prover. Given a trigger p, $\mathcal{M}(p)$ is a set of tuples (t,σ,μ) where $t \in \Gamma$, σ is a substitution from $\mathcal{V}(p)$ to Γ, and μ a type substitution from $\mathcal{TV}(p)$ to ground types, such that $p[\mu]\sigma =_\Delta t$. The computation of $\mathcal{M}(p)$ can be recursively defined on the structure of the trigger p, in the following way:

$$\begin{aligned}
\mathcal{M}(x^{\tau[\bar{\alpha}]}) &= \{(t, \{x \mapsto t\}, \{\bar{\alpha} \mapsto \bar{\tau}\}) \mid t : \tau[\bar{\tau}] \in \Gamma\} \\
\mathcal{M}(c^{\tau[\bar{\alpha}]}) &= \{(c_{[\bar{\tau}]}, \{\}, \{\bar{\alpha} \mapsto \bar{\tau}\}) \mid c_{[\bar{\tau}]} : \tau[\bar{\tau}] \in \Gamma\} \\
\mathcal{M}(f(p_1,\ldots,p_n)) &= \left\{ (t, \biguplus \sigma_i, \biguplus \mu_i) \;\middle|\; \begin{array}{c} t = f(t_1,\ldots,t_n) \in \Gamma \\ \forall i, (t_i, \sigma_i, \mu_i) \in \mathcal{M}(p_i) \\ \sigma_1,\ldots,\sigma_n \text{ } \Delta\text{-compatible} \\ \mu_1,\ldots,\mu_n \text{ compatible} \end{array} \right\}
\end{aligned}$$

It can be shown that if $(t,\sigma,\mu) \in \mathcal{M}(p)$, all variables in $\mathcal{V}(p)$ are instantiated in σ and similarly, all type variables in $\mathcal{TV}(p)$ are instantiated in μ. In other words, t is indeed a ground and monomorphic instance of the original pattern p. This definition is not computationally efficient and in an actual implementation, better strategies can be used:

- when matching a pattern $f(p_1,\ldots,p_n)$, it is possible to retrieve all the terms of the form $f(t_1,\ldots,t_n)$ in Γ and to match the sub-patterns p_i only against the relevant t_i;
- the substitutions can be built incrementally through the matching process in order to only check compatibility at the variables' level and to avoid the cost of merging substitutions.

Finally, it is straightforward to extend our definition of matching to the case of *multi-triggers*, by considering each trigger in turn and discarding incompatible substitutions.

5 Discussion and Conclusion

So far, we have seen how polymorphism had been added to the logic of our SMT solver Alt-Ergo. It is worth noting that, as argued in Section 2, provided that the instantiation mechanism can generate the monomorphic instances of polymorphic axioms, the task of solving a particular problem comes down to solving a monomorphic problem. The important consequence of this remark is that no other modification to our solver was required: in particular, the congruence closure mechanism, the SAT solver or the built-in decision procedures for theories such as linear arithmetic needed not be changed since they would always be used on ground monomorphic terms and formulas. Another consequence is that our approach does not require the combination of parametric theories as presented in [7].

Although it seems reasonably easy to add parametric polymorphism to an SMT solver, one may wonder if it is up to a solver to manipulate logics with complicated features. Indeed, SMT solvers are typically used to certify problems that have been automatically generated by systems such as *verification condition generators* (VCG). It could be argued that even if more refined logics are desirable in such systems (so as to achieve more expressiveness and more compact specifications of programs), the additional burden should be borne by VCG themselves. We have actually studied and used such alternatives in order to employ multi sorted or unsorted provers with the Why toolkit [5]. In [2], the authors show that the problem of encoding polymorphic first-order logic in a weaker logic is not an easy one: intuitive solutions happen to be either impractical, or incorrect. They also show that there is no evidence that it is always possible to statically generate all the monomorphic instances necessary to solve a problem. The encodings they propose provide a better solution, but still have some shortcomings: less easy to implement, they generate formulas that are bigger than the original ones, and require additional lemmas to smoothly deal with built-in decision procedures. All in all, this results in a loss of efficiency on the solver's side.

We have presented our experience of implementing polymorphism in our SMT solver Alt-Ergo. We showed that adding parametric types to the logic required changing the typing system and the matching mechanism in order to account for

type variables. Because the extent of these modifications is relatively moderate, we are convinced that they are worth the extra cost in design and engineering. While not suffering from the same disadvantages as other solutions such as encodings, they bring the expressiveness and conciseness of polymorphism over to the SMT solver. For that reason, we hope that in the future, polymorphism could be added to the SMT-LIB standard.

References

[1] F. Baader and W. Snyder. Unification Theory. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 8, pages 447–533. Elsevier Science, 2001.

[2] J.-F. Couchot and S. Lescuyer. Handling polymorphism in automated deduction. In *21th International Conference on Automated Deduction (CADE-21)*, volume 4603 of *LNCS (LNAI)*, pages 263–278, Bremen, Germany, July 2007.

[3] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[4] J.-C. Filliâtre and C. Marché. Multi-prover verification of C programs. In *ICFEM*, pages 15–29, 2004.

[5] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In W. Damm and H. Hermanns, editors, *19th International Conference on Computer Aided Verification*, Berlin, Germany, July 2007.

[6] T. Hubert and C. Marché. Separation analysis for deductive verification. In *Heap Analysis and Verification (HAV'07)*, Braga, Portugal, Mar. 2007. `http://www.lri.fr/~marche/hubert07hav.pdf`.

[7] S. Krstić, A. Goel, J. Grundy, and C. Tinelli. Combined satisfiability modulo parametric theories. In O. Grumberg and M. Huth, editors, *TACAS*, volume 4424 of *Lecture Notes in Computer Science*, pages 618–631. Springer, 2007.

[8] D. Le Botlan and D. Rémy. MLF: Raising ML to the power of System F. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 27–38, Aug. 2003.

[9] K. R. M. Leino and A. Wallenburg. Class-local object invariants. In *ISEC '08: Proceedings of the 1st conference on India software engineering conference*, pages 57–66, New York, NY, USA, 2008. ACM.

[10] R. Milner. A theory of type polymorphismn programming. *J. Comput. Syst. Sci.*, 17, 1978.

[11] B. C. Pierce. *Types and Programming Languages*, chapter V. MIT Press, 2002.

[12] S. Ranise and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `http://www.smtcomp.org`, 2006.

[13] H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In M. Hofmann and M. Felleisen, editors, *POPL*, pages 97–108, Nice, France, Jan. 2007.

Cover Algorithms and Their Combination

Sumit Gulwani and Madan Musuvathi

Microsoft Research, Redmond, WA, 98052
{sumitg,madanm}@microsoft.com

Abstract. This paper defines the cover of a formula ϕ with respect to a set of variables V in theory T to be the strongest quantifier-free formula that is implied by $\exists V : \phi$ in theory T. Cover exists for several useful theories, including those that do not admit quantifier elimination. This paper describes cover algorithms for the theories of uninterpreted functions and linear arithmetic. In addition, the paper provides a combination algorithm to combine the cover operations for theories that satisfy some general condition. This combination algorithm can be used to compute the cover a formula in the combined theory of uninterpreted functions and linear arithmetic. This paper motivates the study of cover by describing its applications in program analysis and verification techniques, like symbolic model checking and abstract interpretation.

S. Drossopoulou (Ed.): ESOP 2008, LNCS 4960, pp. 193–207, 2008.

SAT Modulo the Theory of Linear Arithmetic: Exact, Inexact and Commercial Solvers (Presentation-only paper)

Germain Faure, Robert Nieuwenhuis, Albert Oliveras and Enric Rodríguez-Carbonell*

Abstract. Many highly sophisticated tools exist for solving linear arithmetic optimization and feasibility problems. Here we analyze why it is difficult to use these tools inside systems for SAT Modulo Theories (SMT) for linear arithmetic: one needs support for disequalities, strict inequalities and, more importantly, for dealing with incorrect results due to the internal use of imprecise floating-point arithmetic. We explain how these problems can be overcome by means of result checking and error recovery policies.

Second, by means of carefully designed experiments with, among other tools, the newest version of ILOG CPLEX and our own new Barcelogic T-solver for arithmetic, we show that, interestingly, the cost of result checking is only a small fraction of the total T-solver time.

Third, we report on extensive experiments running exactly the same SMT search using CPLEX and Barcelogic as T-solvers, where CPLEX tends to be slower than Barcelogic. We analyze these at first sight surprising results, explaining why tools such as CPLEX are not very adequate (nor designed) for this kind of relatively small incremental problems.

Finally, we show how our result checking techniques can still be very useful in combination with inexact floating-point-based T-solvers designed for incremental SMT problems.

* Tech. Univ. of Catalonia, Barcelona. All authors partially supported by Spanish Min. of Educ. and Science through the LogicTools-2 project, TIN2007-68093-C02-01.

An Algebraic Approach for the Unsatisfiability of Nonlinear Constraints*

Ashish Tiwari

SRI International,
333 Ravenswood Ave,
Menlo Park, CA, U.S.A
tiwari@csl.sri.com

Abstract. We describe a simple algebraic semi-decision procedure for detecting unsatisfiability of a (quantifier-free) conjunction of nonlinear equalities and inequalities. The procedure consists of Gröbner basis computation plus extension rules that introduce new definitions, and hence it can be described as a critical-pair completion-based logical procedure. This procedure is shown to be sound and refutationally complete. When projected onto the linear case, our procedure reduces to the Simplex method for solving linear constraints. If only finitely many new definitions are introduced, then the procedure is also terminating. Such terminating, but potentially incomplete, procedures are used in "incompleteness-tolerant" applications.

* Research supported in part by the National Science Foundation under grants CCR-0311348 and ITR-CCR-0326540.

Using an SMT Solver and Craig Interpolation to Detect and Remove Redundant Linear Constraints in Representations of Non-Convex Polyhedra

Christoph Scholl, Stefan Disch, Florian Pigorsch, Stefan Kupferschmid

Albert-Ludwigs-Universität Freiburg
Georges-Köhler-Allee 51, 79110 Freiburg, Germany

Abstract. We present a method which computes optimized representations for non-convex polyhedra. Our method detects so-called redundant linear constraints in these representations by using an incremental SMT solver and then removes the redundant constraints based on Craig interpolation. The approach is evaluated both for formulas from the model checking context including boolean combinations of linear constraints and boolean variables and for random trees composed of quantifiers, AND-, OR-, NOT-operators, and linear constraints produced by a generator. The results clearly show the advantages of our approach in comparison to state-of-the-art solvers.

1 Introduction

In this paper we present an approach which uses SMT (Satisfiability Modulo Theories) solvers and Craig interpolation [3] for optimizing representations of non-convex polyhedra. Non-convex polyhedra are formed by arbitrary boolean combinations (including conjunction, disjunction and negation) of linear constraints. Non-convex polyhedra have been used to represent sets of states of hybrid systems. Whereas approaches like [12, 11] consider unions of convex polyhedra (i.e. unions of conjunctions of linear constraints) together with an explicit representation of discrete states, in [5, 4] a data structure called LinAIGs was used as a single symbolic representation for sets of states of hybrid systems with large discrete state spaces (in the context of model checking by backward analysis). LinAIGs in turn represent an extension of non-convex polyhedra by additional boolean variables, i.e. they represent arbitrary boolean combinations of boolean variables and linear constraints.

In particular, our optimization methods for non-convex polyhedra remove so-called *redundant linear constraints* from our representations. A linear constraint is called redundant for a non-convex polyhedron if and only if the non-convex polyhedron can be described without using this linear constraint. Note that an alternative representation of the polyhedron without using the redundant linear constraint may require a completely different boolean combination of linear constraints. In that sense our method significantly extends results for eliminating redundant linear constraints from convex polyhedra used by Frehse [11] and

Wang [22].[1] Removing redundant linear constraints from non-convex polyhedra plays an important role especially during the elimination of quantifiers for real-valued variables in the context of model checking for hybrid systems. Previous work [4] demonstrated how already a simple preliminary version of redundancy removal can be used during Weispfennig–Loos quantifier elimination [14]: Based on

- the fact that the size of the formula produced by Weispfennig–Loos quantifier elimination of *one* real variable may grow by a factor which is linear in the number of constraints in the original formula and
- the observation that a large number of the new linear constraints which were generated during quantifier elimination was in fact redundant

we were able to show that it is essential to make use of redundancy removal (keeping the number of linear constraints in our representations as small as possible) in order to enable sequences of quantifier eliminations during model checking of non-trivial examples.

Our paper makes the following contributions:

- We present an algorithm for detecting a maximal number of linear constraints which can be removed *simultaneously*. The algorithm is based on sets of don't cares which result from inconsistent assignments of truth values to linear constraints. We show how the *detection* of sets of redundant constraints can be performed using an SMT solver. In particular we show how to use *incremental* SMT solving for detecting larger and larger sets of redundant constraints until a maximal set is obtained.
- Based on the don't care sets mentioned above we provide a detailed proof showing the correctness of the algorithm.
- We show how the information needed for *removing* redundant linear constraints can be extracted from the conflict clauses of an SMT solver. Finally, we present a novel method really performing the removal of redundant linear constraints based on this information. The method is based on Craig interpolation [3, 18, 15].

It is important to note that our method using redundancy elimination is not only applicable in the context of model checking for hybrid systems, but it provides a general method making quantifier elimination for linear arithmetic more efficient. Therefore our experimental evaluation is not only done for formulas generated during runs of the model checker from [4], but also for formulas from [9, 2] (in SMT-LIB format [19]) which consist of arbitrary boolean combinations of linear constraints, combined with quantifications of real-valued variables. For such formulas we solve two problems: First, we compute whether the resulting formula is satisfiable by any assignment of values to the free variables and secondly we do even more, we also compute a predicate over the free variables which is true for all satisfying assignments of the formula. We compare our results to the results of the automata-based tool LIRA [9, 2] (which also solves both problems mentioned above) and to the results of state-of-the-art SMT solvers Yices [7] and CVC3 [20] (which solve the first problem of checking whether the formula is

[1] For convex polyhedra redundancy of linear constraints reduces to the question whether the linear constraint can be omitted in the conjunction of linear constraints without changing the represented set.

satisfiable). Whereas these solvers are not restricted to the subclass of formulas we consider in this paper (and are not optimized for this subclass in the case of Yices and CVC3), our experiments show that for the subclass of formulas considered here our method is much more effective. Our results are obtained by an elaborate scheme combining several methods for keeping representations of intermediate results compact with redundancy removal as an essential component. Internally, these methods make heavy use of the results of SMT solvers restricted to quantifier-free satisfiability solving.[2] Our results suggest to make use of our approach, if the formula at hand belongs to the subclass of linear arithmetic with quantification over reals and moreover, even for more general formulas, one can imagine to use our method as a fast preprocessor for simplifying subformulas from this subclass.

The paper is organized as follows: In Sect. 2 we give a brief review of our representations of non-convex polyhedra, Craig interpolation, and Weispfennig-Loos quantifier elimination. In Sect. 3 we give a definition of redundant linear constraints and present methods for detecting and removing them from representations of non-convex polyhedra. After presenting our encouraging experimental results in Sect. 4 we conclude the paper in Sect. 5.

2 Preliminaries

2.1 Representation of Non-Convex Polyhedra

We assume disjoint sets of variables C and B. The elements of $C = \{c_1, \ldots, c_f\}$ are continuous variables, which are interpreted over the reals $\mathbb{R}$. The elements of $B = \{b_1, \ldots, b_k\}$ are boolean variables and range over the domain $\mathbb{B} = \{0, 1\}$. When we consider logic formulas over $B \cup C$, we restrict terms over C to the class of linear terms of the form $\sum \alpha_i c_i + \alpha_0$ with rational constants α_i and $c_i \in C$. Predicates are given by the set $\mathcal{L}(C)$ of linear constraints, they have the form $t \sim 0$, where $\sim \in \{=, <, \leq\}$ and t is a linear term. $\mathcal{P}(C)$ is the set of all boolean combinations of linear constraints over C, the formulas from $\mathcal{P}(C)$ represent *non-convex polyhedra* over $\mathbb{R}^f$. In this paper we consider the class of formulas from $\mathcal{P}(B, C)$ which is the set of all boolean combinations of boolean variables from B and linear constraints over C.

As a underlying data structure for our method we use representations of formulas from $\mathcal{P}(B, C)$ by LinAIGs [5, 4]. LinAIGs are And-Inverter-Graphs (AIGs) enriched by linear constraints. The structure of LinAIGs is illustrated in Fig. 1.

The component of LinAIGs representing boolean formulas consists in a variant of AIGs, the so–called Functionally Reduced AND-Inverter Graphs (FRAIGs) [16, 17]. AIGs enjoy a widespread application in combinational equivalence checking and Bounded Model Checking (BMC). They are basically boolean circuits consisting only of AND gates and inverters. In [17] FRAIGs were tailored towards the representation and manipulation of sets of states in symbolic model checking, replacing BDDs as a compact representation of large discrete state spaces.

In LinAIGs (see Fig. 1) we use a set of new (boolean) *constraint variables* Q as encodings for the linear constraints, where each occurring $\ell_i \in \mathcal{L}(C)$ is encoded by some $q_{\ell_i} \in Q$. In order to keep the representation compact, we avoid to

[2] In our implementation we use Yices [7] and HySAT [10] for this task.

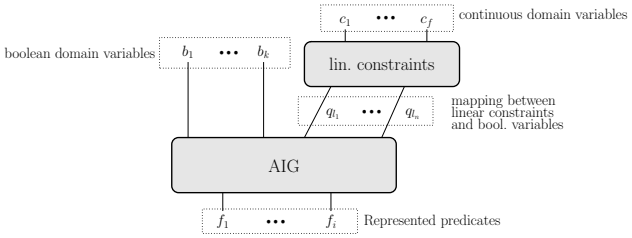

Fig. 1. The LinAIG structure

represent equivalent predicates by different LinAIG nodes. Basically, this could be achieved by checking all pairs of nodes for equivalence (taking the interpretation of constraint variables q_{ℓ_i} by the corresponding linear constraints ℓ_i into account). This check can be performed by an SMT (SAT modulo theories) solver which combines DPLL with linear programming as a decision procedure. Instead of using SMT solver calls for all pairs of nodes, we make use of a carefully designed and tested strategy which avoids SMT solver calls when non-equivalence can be shown using test vectors with valuations $\mathbf{c} \in \mathbb{R}^f$ or when equivalence can be proven already for the boolean abstraction without referring to the definition of the constraint variables. (In this context boolean reasoning is supported by (approximate) knowledge on linear constraints like implications between constraints.)

2.2 Craig interpolation

As we will describe in Sect. 3.3 we remove linear constraints which were found to be redundant using Craig interpolation [3, 18]. Recently, Craig interpolation was applied by McMillan for generating an overapproximated image operator to be used in connection with Bounded Model Checking [15] or by Lee et al. for computing a so-called dependency function in logic synthesis algorithms [13]. A Craig interpolant is computed for a boolean formula F in Conjunctive Normal Form (CNF) (i.e. for a conjunction of disjunctions of boolean variables) which is partitioned into two parts A and B with $F = A \wedge B$. When $F = A \wedge B$ is unsatisfiable, a Craig interpolant for the pair (A, B) is a boolean formula P with the following properties:

- A implies P,
- $P \wedge B$ is unsatisfiable, and
- P depends only on common variables of A and B.

An appropriate Craig interpolant P can be computed based on a proof by resolution that F is unsatisfiable (time and space for this are linear in the size of the proof) [18, 15]. Proofs of unsatisfiability can be computed by any modern SAT solver with proof logging turned on.

2.3 Quantifier elimination

In [4] Loos's and Weispfenning's test point method [14] was adapted to the LinAIG data structure described above. The method eliminates universal quantifiers by converting them into finite conjunctions and existential quantifiers by

converting them into finite disjunctions. The subformulas to be combined by conjunction (or disjunction in case of existential quantification) are obtained from the original formula by replacing real-valued variables by appropriate 'test points' arriving again at formulas in linear arithmetic. The test point method is well-suited for our LinAIG data structure, since substitutions and disjunctions / conjunctions can be performed efficiently in the LinAIG package and the method does not need (potentially costly) conversions of the original formula into CNF / DNF before applying quantifier elimination as the Fourier-Motzkin algorithm, e.g..

The number of test points needed depends linearly on the number of linear constraints in the original formula. Thus, during elimination of one real-valued variable, the number of linear constraints may grow quadratically with the number of linear constraints in the original formula. For sequences of quantifier eliminations it is therefore important to keep the number of linear constraints as small as possible. For this reason we developed an algorithm which computes representations depending on a minimal set of linear constraints. The method is presented in Sect. 3. Experimental results in Sect. 4 show that the method is indeed essential in order to enable sequences of quantifier eliminations.

3 Redundant Linear Constraints

In this section we present our methods to detect and remove redundant linear constraints from non-convex polyhedra.

For illustration of redundant linear constraints see Fig. 2 and 3, which show a typical example stemming from a model checking application. It represents a small state set based on two real variables: Lines in Figures 2 and 3 represent linear constraints, and the gray shaded area represents the space defined by some boolean combination of these constraints. Whereas the representation depicted in Fig. 2 contains 24 linear constraints, a closer analysis shows that an optimized representation can be found using only 15 linear constraints as depicted in Fig. 3.

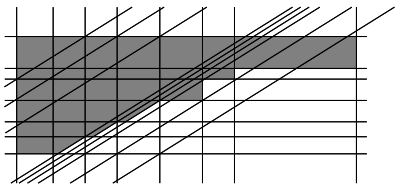

Fig. 2. Before redundancy removal

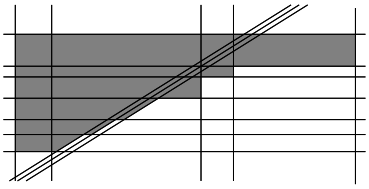

Fig. 3. After redundancy removal

3.1 Redundancy Detection and Removal for Convex Polyhedra

Note that our redundancy detection and removal approach works for representations of *non-convex* polyhedra. Therefore the task is not as straightforward as for other approaches such as [12, 11] which represent sets of *convex* polyhedra, i. e., sets of conjunctions $\ell_1 \wedge \ldots \wedge \ell_n$ of linear constraints. If one is restricted to convex polyhedra, the question whether a linear constraint ℓ_1 is redundant in the representation reduces to the question whether $\ell_2 \wedge \ldots \wedge \ell_n$ represents the same

polyhedron as $\ell_1 \wedge \ldots \wedge \ell_n$, or equivalently, whether $\overline{\ell_1} \wedge \ell_2 \wedge \ldots \wedge \ell_n$ represents the empty set. This question can simply be answered by a linear program solver.

3.2 Detection of Redundant Constraints for Non-convex Polyhedra

Now we consider the case of non-convex polyhedra. To be more precise, we actually consider the slightly generalized case of boolean combinations of linear constraints and additional boolean variables instead of non-convex polyhedra. Our approach works (regardless of the boolean variables) for predicates $F(\ell_1, \ldots, \ell_n, b_1, \ldots, b_k)$ where $\ell_1, \ldots, \ell_n$ are linear constraints over C, $b_1, \ldots, b_k$ are boolean variables, and F is an arbitrary boolean function. Such predicates may be represented by LinAIGs, e.g..

Definition 1 (Redundancy of linear constraints). *The linear constraints $\ell_1, \ldots, \ell_r$ ($1 \leq r \leq n$) are called redundant in the representation of $F(\ell_1, \ldots, \ell_n, b_1, \ldots, b_k)$ iff there is a boolean function G with the property that $F(\ell_1, \ldots, \ell_n, b_1, \ldots, b_k)$ and $G(\ell_{r+1}, \ldots, \ell_n, b_1, \ldots, b_k)$ represent the same predicates.*

In the following we will first prove a theorem which gives a necessary and sufficient condition for a subset $\{\ell_1, \ldots, \ell_r\}$ of linear constraints to be redundant, then we will present an algorithm based on incremental SMT solving which constructs a maximal set of redundant constraints, and finally, we develop a method really computing a representation not depending on $\{\ell_1, \ldots, \ell_r\}$ assuming that the redundancy check was successful.

In order to be able to check for redundancy, we assume a disjoint copy $C' = \{c'_1, \ldots, c'_f\}$ of the continuous variables $C = \{c_1, \ldots, c_f\}$. Moreover, for each linear constraint ℓ_i ($1 \leq i \leq n$) we introduce a corresponding linear constraint ℓ'_i which coincides with ℓ_i up to replacement of variables $c_j \in C$ by variables $c'_j \in C'$. Our check for redundancy is based on the following theorem:

Theorem 1 (Redundancy check). *The linear constraints $\ell_1, \ldots, \ell_r$ ($1 \leq r \leq n$) are redundant in the representation of $F(\ell_1, \ldots, \ell_n, b_1, \ldots, b_k)$ if and only if the predicate*

$$F(\ell_1, \ldots, \ell_n, b_1, \ldots, b_k) \oplus F(\ell'_1, \ldots, \ell'_n, b_1, \ldots, b_k) \wedge \textstyle\bigwedge_{i=r+1}^{n} (\ell_i \equiv \ell'_i) \qquad (1)$$

(where $\oplus$ denotes exclusive-or and $\equiv$ denotes boolean equivalence) is not satisfiable by any assignment of real values to the variables $c_1, \ldots, c_f$, $c'_1, \ldots, c'_f$ and of boolean values to $b_1, \ldots, b_k$.

Note that the check from Thm. 1 can be performed by a (conventional) SMT solver.

Proof (Proof of Thm. 1, only-if-part).

For the proof of the 'only-if-part' of Thm. 1 we assume that the predicate from formula (1) is satisfiable and under this assumption we prove that it cannot be the case that all linear constraints $\ell_1, \ldots, \ell_r$ are redundant, i.e., that there is no boolean function G such that $G(\ell_{r+1}, \ldots, \ell_n, b_1, \ldots, b_k)$ and $F(\ell_1, \ldots, \ell_n, b_1, \ldots, b_k)$ represent the same predicates.

Now consider some satisfying assignment to the predicate from formula (1) as follows: For the real variables $c_1 := v_{c_1}, \ldots, c_f := v_{c_f}$ with $(v_{c_1}, \ldots, v_{c_f}) \in \mathbb{R}^f$,

for the copied real variables $c'_1 := v_{c'_1}, \ldots, c'_f := v_{c'_f}$ with $(v_{c'_1}, \ldots, v_{c'_f}) \in \mathbb{R}^f$, and for the boolean variables $b_1 := v_{b_1}, \ldots, b_k := v_{b_k}$ with $(v_{b_1}, \ldots, v_{b_k}) \in \{0,1\}^k$.

This satisfying assignment implies a corresponding truth assignment to the linear constraints by $\ell_i(v_{c_1}, \ldots, v_{c_f}) = v_{\ell_i}$ $(1 \leq i \leq n)$ with $v_{\ell_i} \in \{0,1\}$ and to the copied linear constraints by $\ell'_i(v_{c'_1}, \ldots, v_{c'_f}) = v_{\ell'_i}$ $(1 \leq i \leq n)$ with $v_{\ell'_i} \in \{0,1\}$.

Since the assignment satisfies formula (1), it holds that

$$F(v_{\ell_1}, \ldots, v_{\ell_n}, v_{b_1}, \ldots, v_{b_k}) \neq F(v_{\ell'_1}, \ldots, v_{\ell'_n}, v_{b_1}, \ldots, v_{b_k}), \quad (a)$$
$$v_{\ell_i} = v_{\ell'_i} \text{ for all } r+1 \leq i \leq n. \quad (b)$$

(Part (a) holds because of the first part of formula (1), i. e. $F(b_1, \ldots, b_k, \ell_1, \ldots, \ell_n) \oplus F(b_1, \ldots, b_k, \ell'_1, \ldots, \ell'_n)$, and part (b) holds because of the second part $\bigwedge_{i=r+1}^{n}(\ell_i \equiv \ell'_i)$.)

Thus the satisfying assignment produces two assignments to the inputs of F which may differ only in the first r values, whereas the function values of F for these two assignments differ. However, any boolean function G not depending on the first r inputs cannot see the difference between these two assignments and thus, it cannot produce different outputs for these two assignments as F. Thus, it is clear that any G not depending on the first r inputs cannot represent the same predicate as F.

For the 'if-part' of Thm. 1 it remains to be shown that an appropriate function G can be constructed, if formula (1) is unsatisfiable.

When constructing G, we need the notion of the *don't care set DC induced by linear constraints*:

Definition 2. *The don't care set DC induced by linear constraints* $\ell_1, \ldots, \ell_n$ *is defined as* $DC := \{(v_{\ell_1}, \ldots, v_{\ell_n}, v_{b_1}, \ldots, v_{b_k}) \mid \nexists (v_{c_1}, \ldots, v_{c_f}) \in \mathbb{R}^f$ *with* $\ell_i(v_{c_1}, \ldots, v_{c_f}) = v_{\ell_i} \forall 1 \leq i \leq n, (v_{b_1}, \ldots, v_{b_k}) \in \{0,1\}^k\}$.

This don't care set contains all assignments of truth values to linear constraints which are inconsistent in the sense that the corresponding linear constraints cannot hold these truth values at the same time.

Based on the set DC, an appropriate boolean function G can be constructed with $G(\ell_{r+1}, \ldots, \ell_n, b_1, \ldots, b_k)$ and $F(\ell_1, \ldots, \ell_n, b_1, \ldots, b_k)$ representing the same predicates, if there is no satisfying assignment to formula (1). This proves the if-part of the proof for Thm. 1. However, we omit this proof here, since we will give an alternative constructive proof for the if-part in Sect.3.3. This constructive proof makes use of a subset DC' of DC which is computed by an SMT solver during the proof of unsatisfiability for formula (1).

Overall algorithm for redundancy detection Now we can present our overall algorithm detecting a maximal set of linear constraints which can be removed from the representation *at the same time*. We start with a small example demonstrating the effect that it is not enough to consider redundancy of single linear constraints and to construct larger sets of redundant constraints simply as unions of smaller sets.

Example 1. Consider the predicate $F(c_1, c_2) = (c_1 \geq 0) \wedge (c_2 \geq 0) \wedge \neg(c_1 + c_2 \leq 0) \wedge \neg(2c_1 + c_2 \leq 0)$. It is easy to see that both the third and the forth linear

constraint in the conjunction have the effect of 'removing the value $(c_1, c_2) = (0, 0)$ from the predicate $F'(c_1, c_2) = (c_1 \geq 0) \wedge (c_2 \geq 0)$'. Therefore both $\ell_3 = (c_1 + c_2 \leq 0)$ and $\ell_4 = (2c_1 + c_2 \leq 0)$ are obviously redundant linear constraints in F. However, it is easy to see that ℓ_3 and ℓ_4 are not redundant in the representation of F **at the same time**, i.e., only $\neg(c_1 + c_2 \leq 0)$ **or** $\neg(2c_1 + c_2 \leq 0)$ can be omitted in the representation for F.

This observation motivates the following overall algorithm to detect a maximal set of redundant linear constraints:

Input : Predicate $F(\ell_1, \ldots, \ell_n, b_1, \ldots, b_k)$
Output: S: Maximal set of redundant linear constraints
begin
  $S := \emptyset$;
  for $i := 1$ **to** n **do**
    if *redundant*$(F, S \cup \{\ell_i\})$ **then**
      $S := S \cup \{\ell_i\}$;
  return S;
end

redundant$(F, S \cup \{l_i\})$ implements the check from Thm. 1 by using an SMT solver. It is important to note that the n SMT problems to be solved in the above loop share almost all of their clauses. For that reason we make use of an *incremental* SMT solver to solve this series of problems. An incremental SMT solver is able to profit from the similarity of the problems by transferring learned knowledge from one SMT solver call to the next (by means of learned conflict clauses). Experimental results in Sect. 4 indeed show the advantage of using an incremental SMT solver.

3.3 Removal of Redundant Linear Constraints for Non-convex Polyhedra

Suppose that the linear constraints $\ell_1, \ldots, \ell_r$ are redundant in $F(\ell_1, \ldots, \ell_n, b_1, \ldots, b_k)$. Now we are looking for an efficient procedure to compute a boolean function G which is appropriate in the sense of Def. 1 and does not depend on the first r inputs. As already mentioned in Sect. 3.2 it is possible to define a method which in principle computes such a function G using the don't care set DC (according to Def. 2). However, an efficient realization of this method would certainly need a compact representation of the don't care set DC. Fortunately, a closer look at the problem reveals the following two interesting observations which turn the basic idea into a feasible approach:

1. In general, we do not need the complete set DC for the computation of the boolean function G.
2. A representation of a sufficient subset of DC which is needed for removing the redundant constraints $\ell_1, \ldots, \ell_r$ is already computed by an SMT solver when checking the satisfiability of formula (1), if one assumes that the SMT solver uses the option of minimizing conflict clauses.

In order to explain how an appropriate subset of DC is computed by the SMT solver (when checking the satisfiability of formula (1)) we need to have a closer look at the functionality of an SMT solver:

An SMT solver introduces constraint variables q_{ℓ_i} for linear constraints ℓ_i (just as in LinAIGs as shown in Fig. 1). First, the SMT solver looks for satisfying assignments to the boolean variables (including the constraint variables). Whenever the SMT solver detects a satisfying assignment to the boolean variables, it checks whether the assignment to the constraint variables is consistent, i. e., whether it can be produced by replacing real-valued variables by reals in the linear constraints. This task is performed by a linear program solver. If the assignment is consistent, then the SMT solver has found a satisfying assignment, otherwise it continues searching for satisfying assignments to the boolean variables. If some assignment $\epsilon_1, \ldots, \epsilon_m$ to constraint variables $q_{\ell_{i_1}}, \ldots, q_{\ell_{i_m}}$ was found to be inconsistent, then the boolean 'conflict clause' $(q_{\ell_{i_1}}^{\overline{\epsilon_1}} + \ldots + q_{\ell_{i_m}}^{\overline{\epsilon_m}})$ is added to the set of clauses in the SMT solver to avoid running into the same conflict again. The negation of this conflict clause describes a set of don't cares due to an inconsistency of linear constraints.

Now consider formula (1) which has to be solved by an SMT solver and suppose that the solver introduces boolean constraint variables q_{ℓ_i} for linear constraints ℓ_i and $q_{\ell'_i}$ for ℓ'_i $(1 \leq i \leq n)$. Since linear constraints $\ell_1, \ldots, \ell_r$ are redundant, formula (1) is unsatisfiable (see Thm. 1). This means that whenever there is some satisfying assignment to boolean variables (including constraint variables) in the SMT solver, it will be necessarily shown to be inconsistent. The most important observation is now that the negations of conflict clauses due to these inconsistencies include the don't cares needed to compute an appropriate boolean function G.

In order to see this, we define for arbitrary values $(v_{\ell_{r+1}}, \ldots, v_{\ell_n}, v_{b_1}, \ldots, v_{b_k}) \in \{0,1\}^{n-r+k}$ the sets $\mathit{orbit}(v_{\ell_{r+1}}, \ldots, v_{\ell_n}, v_{b_1}, \ldots, v_{b_k}) := \{(v_{\ell_1}, \ldots, v_{\ell_r}, v_{\ell_{r+1}}, \ldots, v_{\ell_n}, v_{b_1}, \ldots, v_{b_k}) \mid (v_{\ell_1}, \ldots, v_{\ell_r}) \in \{0,1\}^r\}$.

If there is an orbit $\mathit{orbit}(v_{\ell_{r+1}}, \ldots, v_{\ell_n}, v_{b_1}, \ldots, v_{b_k})$ containing two different elements $v^{(1)} := (v_{\ell_1}, \ldots, v_{\ell_r}, v_{\ell_{r+1}}, \ldots, v_{\ell_n}, v_{b_1}, \ldots, v_{b_k})$ and $v^{(2)} := (v'_{\ell_1}, \ldots, v'_{\ell_r}, v_{\ell_{r+1}}, \ldots, v_{\ell_n}, v_{b_1}, \ldots, v_{b_k})$ with $F(v^{(1)}) \neq F(v^{(2)})$, then the following assignment to the boolean variables obviously satisfies the boolean abstraction of formula (1) in the SMT solver: $q_{\ell_1} := v_{\ell_1}, \ldots, q_{\ell_r} := v_{\ell_r}, q_{\ell'_1} := v'_{\ell_1}, \ldots, q_{\ell'_r} := v'_{\ell_r}$, $q_{\ell_{r+1}} := q_{\ell'_{r+1}} := v_{\ell_{r+1}}, \ldots, q_{\ell_n} := q_{\ell'_n} := v_{\ell_n}$, $b_1 := v_{b_1}, \ldots, b_k := v_{b_k}$.

Since we assumed that formula (1) is not satisfiable, this assignment cannot be consistent wrt. the interpretation of constraint variables by linear constraints. So there must be an inconsistency in the truth assignment to some linear constraints $\ell_1, \ldots, \ell_n, \ell'_1, \ldots, \ell'_n$. Since the linear constraints ℓ_i and ℓ'_i are based on disjoint sets of real variables $C = \{c_1, \ldots, c_f\}$ and $C' = \{c'_1, \ldots, c'_f\}$, respectively, it is easy to see that a minimal number of assignments which are already inconsistent contains *either* only assignments to a subset of $\ell_1, \ldots, \ell_n$ *or* to a subset of $\ell'_1, \ldots, \ell'_n$.[3] When using the option of minimizing conflict clauses, the SMT solver will thus learn a conflict clause whose negation either contains the don't care $v^{(1)} = (v_{\ell_1}, \ldots, v_{\ell_r}, v_{\ell_{r+1}}, \ldots, v_{\ell_n}, v_{b_1}, \ldots, v_{b_k})$ or the don't care $v^{(2)} = (v'_{\ell_1}, \ldots, v'_{\ell_r}, v_{\ell_{r+1}}, \ldots, v_{\ell_n}, v_{b_1}, \ldots, v_{b_k})$. Since this consideration holds for

[3] For our purposes, it does not matter whether the inconsistency is given in terms of linear constraints $\ell_1, \ldots, \ell_n$ or $\ell'_1, \ldots, \ell'_n$. We are only interested in assignments of boolean values to linear constraints leading to inconsistencies; of course, the same inconsistencies will hold both for $\ell_1, \ldots, \ell_n$ and their copies $\ell'_1, \ldots, \ell'_n$.

all pairs of elements in some orbit $orbit(v_{\ell_{r+1}}, \ldots, v_{\ell_n}, v_{b_1}, \ldots, v_{b_k})$ for which F produces different values, this means for the subset $DC' \subseteq DC$ of don't cares detected during the run of the SMT solver: If $orbit(v_{\ell_{r+1}}, \ldots, v_{\ell_n}, v_{b_1}, \ldots, v_{b_k})$ is not completely contained in DC', then $|F(orbit(v_{\ell_{r+1}}, \ldots, v_{\ell_n}, v_{b_1}, \ldots, v_{b_k}) \setminus DC')| = 1$ (or in other words: the elements of $orbit(v_{\ell_{r+1}}, \ldots, v_{\ell_n}, v_{b_1}, \ldots, v_{b_k})$ which are not in DC' are either all mapped by F to 0 or are all mapped by F to 1).

Now we define the function value of G for each $(v_{\ell_{r+1}}, \ldots, v_{\ell_n}, v_{b_1}, \ldots, v_{b_k}) \in \{0,1\}^{n-r+k}$:

1. If $orbit(v_{\ell_{r+1}}, \ldots, v_{\ell_n}, v_{b_1}, \ldots, v_{b_k}) \subseteq DC'$, then $G(v_{\ell_{r+1}}, \ldots, v_{\ell_n}, v_{b_1}, \ldots, v_{b_k})$ is chosen arbitrarily.
2. Otherwise $G(v_{\ell_{r+1}}, \ldots, v_{\ell_n}, v_{b_1}, \ldots, v_{b_k}) = \delta$ with $F(orbit(v_{\ell_{r+1}}, \ldots, v_{\ell_n}, v_{b_1}, \ldots, v_{b_k}) \setminus DC') = \{\delta\}$, $\delta \in \{0,1\}$.

It is easy to see that G does not depend on variables $q_{\ell_1}, \ldots, q_{\ell_r}$ and that G is well-defined (this follows from $|F(orbit(v_{\ell_{r+1}}, \ldots, v_{\ell_n}, v_{b_1}, \ldots, v_{b_k}) \setminus DC')| = 1$), i.e. G is a possible solution according to Def. 1. This consideration also provides a proof for the if-part of Thm. 1. Note that according to case 1) of the definition above there may be several possible choices fulfilling the definition of G.

A predicate dc which describes the don't cares in DC' may be extracted from the SMT solver as a disjunction of negated conflict clauses which record inconsistencies between linear constraints.

Redundancy Removal by Existential Quantification A straightforward way of computing an appropriate function G relies on existential quantification:

- At first by $G' = F \wedge \overline{dc}$ all don't cares represented by dc are mapped to the function value 0.
- Secondly, we perform an existential quantification of the variables $q_{\ell_1}, \ldots, q_{\ell_r}$ in G': $G = \exists q_{\ell_1}, \ldots, q_{\ell_r} G'$. This existential quantification maps all elements of an orbit $orbit(v_{\ell_{r+1}}, \ldots, v_{\ell_n}, v_{b_1}, \ldots, v_{b_k})$ to 1, whenever the orbit contains an element ϵ with $dc(\epsilon) = 0$ and $F(\epsilon) = 1$. Since due to the argumentation above there is no other element δ in such an orbit with $dc(\delta) = 0$ and $F(\delta) = 0$, G eventually differs from F only for don't cares defined by dc and it certainly does not depend on variables $q_{\ell_1}, \ldots, q_{\ell_r}$, i.e. existential quantification computes one possible solution for G according to Def. 1 (more precisely it computes exactly the solution for G which maps a minimum number of elements of $\{0,1\}^{n-r+k}$ to 1).

Redundancy Removal with Craig Interpolants Although our implementation of LinAIGs supports quantification of boolean variables by a series of methods like avoiding the insertion of equivalent nodes (see Sect. 2.1), quantifier scheduling, BDD sweeping and node selection heuristics (see [17]), there remains the risk of doubling the representation size by quantifying a single boolean variable.[4] Therefore the computation of G by $G = \exists q_{\ell_1}, \ldots, q_{\ell_r} G'$ as shown above may potentially lead to large LinAIG representations (although it reduces the number of linear constraints).

[4] Basically, existential quantification of a boolean variable is reduced to a disjunction of both cofactors wrt. 0 and wrt. 1.

On the other hand, this choice for G is only one of many other possible choices. Motivated by these facts we looked for an alternative solution. Here we present a solution which needs only one application of Craig interpolation [3, 18] (see Sect. 2.2) instead of a series of existential quantifications of boolean variables. Note that in this context Craig interpolation leads to an exact result (as one of several possible choices) and not to an approximation as in [15].

Our task is to find a boolean function $G(q_{\ell_{r+1}}, \ldots, q_{\ell_n}, b_1, \ldots, b_k)$ with

$$(F \wedge \overline{dc})(q_{\ell_1}, \ldots, q_{\ell_n}, b_1, \ldots, b_k) \implies G(q_{\ell_{r+1}}, \ldots, q_{\ell_n}, b_1, \ldots, b_k), \quad (2)$$

$$G(q_{\ell_{r+1}}, \ldots, q_{\ell_n}, b_1, \ldots, b_k) \implies (F \vee dc)(q_{\ell_1}, \ldots, q_{\ell_n}, b_1, \ldots, b_k). \quad (3)$$

Now let $A(q_{\ell_1}, \ldots, q_{\ell_r}, q_{\ell_{r+1}}, \ldots, q_{\ell_n}, b_1, \ldots, b_k, h_1, \ldots, h_l)$ represent the CNF for a Tseitin transformation [21] of $(F \wedge \overline{dc})(q_{\ell_1}, \ldots, q_{\ell_r}, q_{\ell_{r+1}}, \ldots, q_{\ell_n}, b_1, \ldots, b_k)$ (with new auxiliary variables $h_1, \ldots, h_l$).

Likewise, let $B(q'_{\ell_1}, \ldots, q'_{\ell_r}, q_{\ell_{r+1}}, \ldots, q_{\ell_n}, b_1, \ldots, b_k, h'_1, \ldots, h'_{l'})$ be the CNF for a Tseitin transformation of $(\overline{F} \wedge \overline{dc})(q'_{\ell_1}, \ldots, q'_{\ell_r}, q_{\ell_{r+1}}, \ldots, q_{\ell_n}, b_1, \ldots, b_k)$ (with new auxiliary variables $h'_1, \ldots, h'_{l'}$ and new copies $q'_{\ell_1}, \ldots, q'_{\ell_r}$ of the variables $q_{\ell_1}, \ldots, q_{\ell_r}$). Then A and B fulfill the precondition for Craig interpolation as given in Sect. 2.2, i.e., $A \wedge B = 0$:

Suppose that there is a satisfying assignment to $A \wedge B$ given by $q_{\ell_1} := v_{\ell_1}, \ldots, q_{\ell_r} := v_{\ell_r}, q'_{\ell_1} := v'_{\ell_1}, \ldots, q'_{\ell_r} := v'_{\ell_r}$, $q_{\ell_{r+1}} := v_{\ell_{r+1}}, \ldots, q_{\ell_n} := v_{\ell_n}$, $b_1 := v_{b_1}, \ldots, b_k := v_{b_k}$ and the corresponding assignments to auxiliary variables $h_1, \ldots, h_l$ and $h'_1, \ldots, h'_{l'}$ which are implied by these assignments. According to the definition of A and B this would mean that the set $\mathit{orbit}(v_{\ell_{r+1}}, \ldots, v_{\ell_n}, v_{b_1}, \ldots, v_{b_k})$ would contain two elements $(v_{\ell_1}, \ldots, v_{\ell_r}, v_{\ell_{r+1}}, \ldots, v_{\ell_n}, v_{b_1}, \ldots, v_{b_k})$ and $(v'_{\ell_1}, \ldots, v'_{\ell_r}, v_{\ell_{r+1}}, \ldots, v_{\ell_n}, v_{b_1}, \ldots, v_{b_k})$ which do not belong to the don't care set DC' and which fulfill $F(v_{\ell_1}, \ldots, v_{\ell_r}, v_{\ell_{r+1}}, \ldots, v_{\ell_n}, v_{b_1}, \ldots, v_{b_k}) = 1$ and $F(v'_{\ell_1}, \ldots, v'_{\ell_r}, v_{\ell_{r+1}}, \ldots, v_{\ell_n}, v_{b_1}, \ldots, v_{b_k}) = 0$. This is a contradiction to the property shown above that the elements of $\mathit{orbit}(v_{\ell_{r+1}}, \ldots, v_{\ell_n}, v_{b_1}, \ldots, v_{b_k})$ which are not in DC' are either all mapped by F to 0 or are all mapped by F to 1.

A Craig interpolant G computed for A and B (e.g. according to [18]) has the following properties:

- It depends only on common variables $q_{\ell_{r+1}}, \ldots, q_{\ell_n}, b_1, \ldots, b_k$ of A and B,
- $A \implies G$, i.e., G fulfills equation (2), and
- $G \wedge B$ is unsatisfiable, or equivalently, $G \implies \overline{B}$, i.e., G fulfills equation (3).

This shows that a Craig interpolant for (A, B) is exactly one of the possible solutions for G which we were looking for.

4 Experimental Results

We implemented redundancy detection by incremental SMT solving and redundancy removal by Craig interpolation in the framework of LinAIGs. The implementation uses two SMT solvers via API calls. Yices [7] is used for all SMT solver calls except the generation of the don't care set. This means that Yices performs all equivalence checks needed for LinAIG compaction as described in Sect. 2.1 and moreover, it is also used for the redundancy detection algorithm described in

Sect. 3 in an incremental way. For the computation of the don't care set required for redundancy removal we use HySAT [10], since we needed an SMT solver where we could modify the source code in order to be able to extract conflict clauses. The computation of the Craig interpolants is done with MiniSAT [8], where we made an extension to the proof logging version. All experiments were performed on an AMD Opteron with 2.6 GHz and 16 GB RAM under Linux.

4.1 Comparison of the LinAIG evolution with and without redundancy removal

In Fig. 4 we present a comparison of two runs of the model checker from [4]. The left diagram shows the evolution of the linear constraints over time and the right diagram shows the evolution of node counts. When we do not use redundancy removal, the number of linear constraint is quickly increasing up to 1000 and more, and the number of AIG nodes is exploding up to a value of 150,000 . On the other hand, when using redundancy removal the number of linear constraints and the number of AIG nodes show only a moderate growth rate. This gives a strong evidence that redundancy removal is absolutely necessary when using quantifier elimination to keep the data structure compact in our model checking environment.

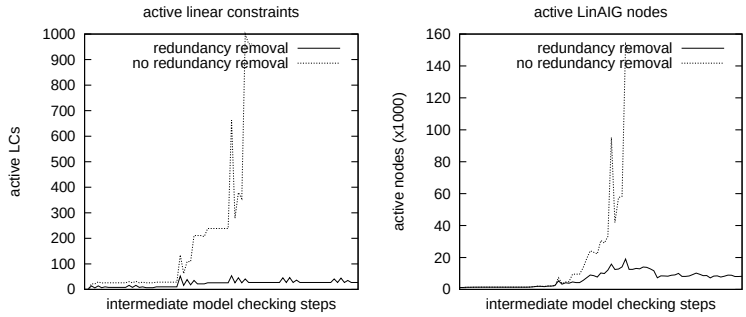

Fig. 4. Comparison of the LinAIG evolution with and without redundancy removal

4.2 Elimination of redundant constraints: Existential quantification vs. Craig interpolation

In a second experiment we compared two different approaches to the removal of redundant constraints as presented in Sect. 3.3. The first one uses existential quantification to eliminate redundant constraints, the second one uses our approach based on Craig interpolation. The benchmarks represent state sets extracted from the model checker in [4] during three runs with different model checking problems, in each case after the elimination of quantifiers over real variables. These problems also contain boolean variables.

The results are given in Table 1. The number of the AIG nodes and linear constraints before redundancy removal are shown in columns 2 and 3. In column 4 the detected number of redundant linear constraints is given. The times for the detection of redundancy and the don't care set generation are given in columns

Table 1. Comparison of redundancy removal: existential quantification vs. Craig interpolants

Benchmark	# AIG nodes	# linear constr.	# redundant lin. constr.	redundancy detection (s)	dc set creation (s)	RR exist. quant. Δ nodes	RR exist. quant. time (s)	RR Craig interp. Δ nodes	RR Craig interp. time (s)
model_1-1	1459	41	22	0.22	1.10	-541	7.19	-814	0.74
model_1-2	1716	74	27	0.51	1.79	-313	13.14	-1047	0.68
model_1-3	2340	105	22	1.89	8.48	459	25.35	-515	2.32
model_1-4	3500	142	28	8.02	41.72	1642	75.49	1062	10.08
model_1-5	2837	123	13	4.61	29.02	-230	12.34	1595	23.10
model_2-1	824	29	8	0.12	0.27	747	2.55	-142	0.43
model_2-2	1424	37	10	0.36	0.80	1104	3.45	233	1.19
model_2-3	3048	52	11	2.45	3.09	1996	10.13	171	4.22
model_2-4	1848	37	14	0.57	1.02	852	4.03	-149	1.34
model_3-1	1775	297	228	1.86	49.28		>7200	-1453	1.60
model_3-2	6703	1281	1143	105.69	2113.20		>7200	-5805	14.12

5 and 6. Note that these values are the same for both approaches, because the difference lies only in the way linear constraints are actually removed. In the last four columns the results of the two algorithms are shown, where 'Δ nodes' denotes the difference between the number of AIG nodes before and after the removal step and 'time' is the CPU time needed for this step. We used a timeout of 7200 seconds and a memory limit of 4 GB.

The results clearly show that wrt. runtime the redundancy removal based on Craig interpolation outperforms the approach with existential quantification by far. Especially when the benchmarks are more complex and show a large number of redundant linear constraints, the difference between the two methods is substantial. Moreover, also the resulting AIG is often much smaller. It is interesting to see that using incremental SMT solving techniques it was in many cases really possible to detect large sets of redundant linear constraints in very short times. As shown in the previous experiment this pays off also in later steps of model checking when quantifier elimination works on a representation with a smaller number of linear constraints. Considering column 6 we observe that runtimes for the generation of don't care sets by HySAT often dominate the overall runtime.[5] For the future we plan to replace HySAT (which is tuned for BMC problems and is clearly outperformed by Yices in our experiments on non-BMC problems) by a state-of-the-art SMT solver allowing the extraction of conflict clauses.[6] This is an issue where we see much room for improvement.

4.3 Comparison of the LinAIG based quantifier elimination vs. other solvers

In order to evaluate our ideas in a more general domain we compared our approach to quantifier elimination with LIRA 1.1.2 [9, 2] which is an automata-based tool capable of representing sets of states over real, integer, and boolean variables and both CVC3 1.2.1 [20] and Yices 1.0.11 [7] which are state-of-the-art SMT solvers. We ran the solvers on three sets of formulas from the class of quantified linear real arithmetic:

[5] As already mentioned above, for technical reasons in our implementation we have to repeat the last step of redundancy detection (which actually was already performed by Yices) using HySAT in order to be able to extract conflict clauses.

[6] This would include also a handling of don't cares which do not occur in the set of (negated) conflict clauses due to 'theory propagation'.

1. model_X: These formulas are representing problems occurring in the model-checker [4] when computing a continuous pre-image of the state set. All formulas of this set contain two quantified variables, one is existentially quantified and the other is universally quantified.
2. RND: These formulas are random trees composed of quantifiers, AND-, OR-, NOT-operators, and linear inequations. The quantifiers are randomly distributed over the whole formula tree. We varied the number of quantified variables and the depth of the trees to get formulas with different difficulty levels. In all cases there was an additional free variable left in the formula. The random benchmarks were generated with the tool also used in [9, 2].
3. RNDPRE: These formulas are similar to the RND set, except that the formulas all consist of a prefix of alternating quantifiers and a quantifier free inner part.

All formulas are given in the SMT-LIB format [19] and are publically available[7].

Since the SMT solvers decide satisfiability of formulas instead of computing predicates representing all satisfying assignments, we interpret free variables as implicitly existentially quantified and decide satisfiability. Both our LinAIG based tool and LIRA additionally compute representations for predicates representing all satisfying assignments. We used a time limit of 1200 CPU seconds and a memory limit of 4 GB.

Table 2 shows the results. The column 'Benchmark' lists the benchmark sets, 'Quantified' lists the number of quantified variables in each formula of the set, 'Instances' shows the number of instances in the set. The columns labeled 'SAT' and 'UNSAT' give the numbers of instances for which the solver returned 'satisfiable' and 'unsatisfiable'. The numbers of instances where the solver returned 'unknown', ran out of memory, or violated the time limit, are listed in the columns 'Unknown', 'Memout', 'Timeout'. Column 'Time (s)' shows the total run times (in CPU seconds) of the solver for the formula set[8], and finally column 'Solved' lists the total numbers of solved instances of the set. The results for CVC3, Yices, LIRA, and our LinAIG based solver using redundancy removal are shown in the column groups labeled 'CVC3', 'Yices','LIRA', and 'LinAIG'.

CVC3 is able to solve 34 out of 380 instances, Yices solves 13 instances. Note however that these solvers are not restricted to the subclass of formulas we consider in this paper. They are able to handle the more general AUFLIRA class of formulas [1] and for handling formulas with quantifiers they make use of heuristics based on E-matching [6] which are not tuned to problems that contain only arithmetic.

The automata-based tool LIRA solves 95 out of 380 instances.

Our experiments show that for the subclass of formulas considered here our method is much more effective: The LinAIG based solver is able to solve 352 out of 380 instances.

5 Conclusions and Future Work

We presented an approach for optimizing non-convex polyhedra based on the removal of redundant constraints. Our experimental results show that our ap-

[7] `http://abs.informatik.uni-freiburg.de/smtbench/`

[8] Unsolved instances (i. e. 'Unknown', 'Memout', and 'Timeout') are considered to contribute 1200 CPU seconds (the time limit)

Table 2. Comparison of Solvers

Benchmark	Quantified	Instances	CVC3							YICES						LIRA						LinAIG				
			SAT	UNSAT	Unknown	Memout	Timeout	Time (s)	**Solved**	SAT	UNSAT	Unknown	Time (s)	**Solved**	SAT	UNSAT	Memout	Timeout	Time (s)	**Solved**	SAT	UNSAT	Timeout	Time (s)	**Solved**
model_4	2	6	0	0	6	0	0	7K	**0**	0	0	6	7K	**0**	2	0	4	0	5K	**2**	6	0	0	11	**6**
model_5	2	64	0	0	64	0	0	77K	**0**	0	0	64	77K	**0**	39	0	12	13	39K	**39**	64	0	0	282	**64**
model_6	2	80	0	0	80	0	0	96K	**0**	0	0	80	96K	**0**	27	0	46	7	69K	**27**	80	0	0	251	**80**
RND	3	30	1	7	21	1	0	26K	**8**	3	3	24	29K	**6**	4	5	21	0	25K	**9**	17	13	0	1K	**30**
RND	4	30	0	2	27	0	1	34K	**2**	1	0	29	35K	**1**	1	1	28	0	34K	**2**	19	10	1	2K	**29**
RND	6	40	0	0	31	7	2	48K	**0**	0	4	36	43K	**4**	1	0	39	0	47K	**1**	18	9	13	16K	**27**
RNDPRE	3	60	0	24	31	4	1	43K	**24**	0	1	59	71K	**1**	6	8	45	1	57K	**14**	34	26	0	2K	**60**
RNDPRE	4	70	0	0	66	4	0	84K	**0**	1	0	69	83K	**1**	1	0	67	2	83K	**1**	28	28	14	21K	**56**
Total		380	1	33	326	16	4	415K	**34**	5	8	367	440K	**13**	81	14	262	23	358K	**95**	266	86	28	43K	**352**

proach can be successfully applied to solving quantified formulas including linear real arithmetic and boolean formulas. Since our method does not only solve satisfiability of formulas, but constructs predicates of all satisfying assignments to the free variables in the formula, our results may suggest to use the presented method in the future also as a fast preprocessor for more general formulas by simplifying subformulas from the subclass considered in this paper. Moreover, it will be interesting to apply the methods to underlying theories different from linear real arithmetic, too.

Acknowledgements

The results presented in this paper were developed in the context of the Transregional Collaborative Research Center 'Automatic Verification and Analysis of Complex Systems' (SFB/TR 14 AVACS) supported by the German Research Council (DFG). We worked in close cooperation with our colleagues from the 'First Order Model Checking team' within subproject H3 and we would like to thank W. Damm, H. Hungar, J. Pang, and B. Wirtz from the University of Oldenburg, and S. Jacobs and U. Waldmann from the Max Planck Institute for Computer Science at Saarbrücken for numerous ideas and helpful discussions. Moreover, we would like to thank Jochen Eisinger from the University of Freiburg for providing the formula generator used in our experiments.

References

1. C. Barrett, M. Deters, A. Oliveras, and A. Stump. Satisfiability Modulo Theories Competition (SMT-COMP) 2008: Rules and Precedures, 2008. `http://smtcomp.org/rules08.pdf`.
2. B. Becker, C. Dax, J. Eisinger, and F. Klaedtke. LIRA: Handling constraints of linear arithmetics over the integers and the reals. In *Proc. of the 19th International Conference on Computer Aided Verification*, 2007, LNCS, pp. 312–315. Springer.
3. W. Craig. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *Journal on Symbolic Logic*, 22(3):269–285, 1957.
4. W. Damm, S. Disch, H. Hungar, S. Jacobs, J. Pang, F. Pigorsch, C. Scholl, U. Waldmann, and B. Wirtz. Exact state set representations in the verification of linear hybrid systems with large discrete state space. In *5th International Symposium*

on Automated Technology for Verification and Analysis, 2007, *LNCS 4762*, pp. 425–440. Springer.
5. W. Damm, S. Disch, H. Hungar, J. Pang, F. Pigorsch, C. Scholl, U. Waldmann, and B. Wirtz. Automatic verification of hybrid systems with large discrete state space. In *4th Symposium on Automated Technology for Verification and Analysis*, 2006, *LNCS 4218*, pp. 276–291.
6. D. Detlefs, G. Nelson, and J. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Systems Research Center, 2003.
7. B. Dutertre and L. de Moura. A fast linear-arithmetic solver for DPLL(T). In *18th Conference on Computer Aided Verification*, 2006, *LNCS 4144*, pp. 81–94. Springer.
8. N. Eén and N. Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing, 6th InternationalConference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8,2003 Selected Revised Papers*, 2003, *LNCS 2919*, pp. 541–638. Springer.
9. J. Eisinger and F. Klaedtke. Don't care words with application to the automata-based approach for real addition. In *Proc. of the 18th International Conference on Computer Aided Verification*, 2006, LNCS, pp. 67–80. Springer.
10. M. Fränzle and C. Herde. HySAT: An efficient proof engine for bounded model checking of hybrid systems. *Formal Methods in System Design*, 30(3):179–198, 2007.
11. G. Frehse. PHAVer: Algorithmic verification of hybrid systems past HyTech. In *8th Workshop on Hybrid Systems: Computation and Control*, 2005, *LNCS 3414*, pp. 258–273. Springer.
12. T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: A model checker for hybrid systems. *Software Tools for Technology Transfer*, 1(1–2):110–122, 1997.
13. C.-C. Lee, J.-H. R. Jiang, C.-Y. Huang, and A. Mishchenko. Scalable exploration of functional dependency by interpolation and incremental SAT solving. In G. G. E. Gielen, ed., *ICCAD*, 2007, pp. 227–233. IEEE.
14. R. Loos and V. Weispfenning. Applying linear quantifier elimination. *The Computer Journal*, 36(5):450–462, 1993.
15. K. L. McMillan. Interpolation and SAT-based model checking. In *15th Conference on Computer Aided Verification*, 2003, *LNCS 2725*, pp. 1–13. Springer.
16. A. Mishchenko, S. Chatterjee, R. Jiang, and R. K. Brayton. FRAIGs: A unifying representation for logic synthesis and verification. Technical report, EECS Dept., UC Berkeley, 2005.
17. F. Pigorsch, C. Scholl, and S. Disch. Advanced unbounded model checking by using AIGs, BDD sweeping and quantifier scheduling. In *6th Conference on Formal Methods in Computer Aided Design*, 2006, pp. 89–96. IEEE Press.
18. P. Pudlák. Lower bounds for resolution and cutting plane proofs and monotone computations. *Journal on Symbolic Logic*, 62(3):981–998, 1997.
19. S. Ranise and C. Tinelli. The SMT-LIB Standard: Version 1.2, 2006. `http://combination.cs.uiowa.edu/smtlib/papers/format-v1.2-r06.08.30.pdf`.
20. A. Stump, C. W. Barrett, and D. L. Dill. CVC: A cooperating validity checker. In *Proceedings of the* 14^{th} *International Conference on Computer Aided Verification*, 2002, *LNCS 2404*, pp. 500–504. Springer.
21. G. Tseitin. On the complexity of derivations in propositional calculus. In A. Slisenko, ed., *Studies in Constructive Mathematics and Mathematical Logics*. 1968.
22. F. Wang. Symbolic parametric safety analysis of linear hybrid systems with BDD-like data-structures. *IEEE Transactions on Software Engineering*, 31(1):38–52, 2005.

Efficient Interpolant Generation in Satisfiability Modulo Theories *

Alessandro Cimatti[1], Alberto Griggio[2], and Roberto Sebastiani[2]

[1] FBK-IRST, Povo, Trento, Italy. cimatti@fbk.eu
[2] DISI, Università di Trento, Italy. {griggio,rseba}@disi.unitn.it

Abstract. The problem of computing Craig Interpolants for propositional (SAT) formulas has recently received a lot of interest, mainly for its applications in formal verification. However, propositional logic is often not expressive enough for representing many interesting verification problems, which can be more naturally addressed in the framework of Satisfiability Modulo Theories, SMT.
Although some works have addressed the topic of generating interpolants in SMT, the techniques and tools that are currently available have some limitations, and their performace still does not exploit the full power of current state-of-the-art SMT solvers.
In this paper we try to close this gap. We present several techniques for interpolant generation in SMT which overcome the limitations of the current generators mentioned above, and which take full advantage of state-of-the-art SMT technology. These novel techniques can lead to substantial performance improvements wrt. the currently available tools.
We support our claims with an extensive experimental evaluation of our implementation of the proposed techniques in the MathSAT SMT solver.

* This work has been partly supported by ORCHID, a project sponsored by Provincia Autonoma di Trento, and by a grant from Intel Corporation.

Rocket-Fast Proof Checking for SMT Solvers

Michał Moskal

University of Wrocław, Poland

Abstract. Modern Satisfiability Modulo Theories (SMT) solvers are used in a wide variety of software and hardware verification applications. Proof producing SMT solvers are very desirable as they increase confidence in the solver and ease debugging/profiling, while allowing for scenarios like Proof-Carrying Code (PCC). However, the size of typical proofs generated by SMT solvers poses a problem for the existing systems, up to the point where proof checking consumes orders of magnitude more computer resources than proof generation. In this paper we show how this problem can be addressed using a simple term rewriting formalism, which is used to encode proofs in a natural deduction style. We formally prove soundness of our rules and evaluate an implementation of the term rewriting engine on a set of proofs generated from industrial benchmarks. The modest memory and CPU time requirements of the implementation allow for proof checking even on a small PDA device, paving a way for PCC on such devices.

C.R. Ramakrishnan and J. Rehof (Eds.): TACAS 2008, LNCS 4963, pp. 486–500, 2008.

Proof Translation and SMT-LIB Benchmark Certification: A Preliminary Report *

Yeting Ge[1], Clark Barrett[1]
[1]New York University, yeting|barrett@cs.nyu.edu

Abstract

Satisfiability Modulo Theories (SMT) solvers are large and complicated pieces of code. As a result, ensuring their correctness is challenging. In this paper, we discuss a technique for ensuring soundness by producing and checking proofs. We give details of our implementation using CVC3 and HOL Light and provide initial results from our effort to certify the SMT-LIB benchmarks.

1 Introduction

Satisfiability Modulo Theories (SMT) solvers have been successfully applied in many verification applications. As modern SMT solvers add optimizations and features, they are becoming more complicated than ever before. At the same time, SMT solvers are increasingly being used in applications where the correctness of the solver is essential. With currently available verification techniques, it would be extremely difficult to verify that a modern SMT solver is correct. Even if such a proof were done, it would be difficult to maintain in the face of constant changes to the solver. One alternative is to have the SMT solver produce a record of its proof search and then use a small, trusted proof-checker to check the proof.

However, the approach of generating and checking proofs from SMT solvers faces several challenges. The first challenge is to design a suitable set of proof rules. Unlike SAT solvers, for which only one proof rule (Boolean resolution) is sufficient for proof-checking, SMT solvers require a much richer set of proof rules, which depend on the background theories supported and the decision procedures employed. There are also trade-offs to be considered in the selection of proof rules. On the one hand, a small set of simple rules is better for proof-checking. On the other hand, a larger set of more complex rules makes things easier for the implementer of the SMT solver. An additional issue is the maintenance of the proof rules. As functionality is

*This work was partially supported by National Science Foundation grant number 0551645.

added and modified over time, proof rules may change and new proof rules may be needed. Adding support for these changes to the proof-checking strategy thus incurs additional maintenance effort.

The next challenge is to implement a proof-checker. Proofs of nontrivial SMT benchmarks are far too big to be readable by a human. Thus, proofs must be checked by a trusted proof-checking algorithm. Depending on the number and complexity of proof rules, the task of a proof-checker may range from fairly simple to very complex. One representation of a proof is as a tree in which each node is labeled with a formula. The root of the tree represents the theorem being proved. Each internal node represents the application of a proof rule used to derive the formula labeling that node from the formulas labeling its children. For such a proof tree, the task of a proof-checker is to check that the deduction represented at each node in the tree is valid. For simple rules, such as deriving $\neg \mathit{true}$ from false, a simple syntactic check is sufficient. However, for more complicated rules, (for example, a single proof rule could be used to encapsulate normalization of linear arithmetic terms), a sophisticated algorithm requiring many steps may be needed.

There seems to be an unavoidable trade-off between performance and ease of coding the SMT solver (which leads to many complex proof rules) and the simplicity of the proof-checker which is desirable in order to minimize the amount of code that must be trusted (and also to minimize the effort required in building and maintaining the checker).

There is, however, a solution that has most of the advantages of both. The idea is to use another existing theorem prover to check proofs from the SMT solver. This approach enables the use of fairly complicated rules in the SMT solver as long as the reasoning behind the rules can be reproduced in the other prover. The additional work that must be done is then to *translate* each rule into the language and methodology of the other theorem prover. A successful check of the proof results in a theorem in the other prover. Notice that we have reduced the problem of trusting the SMT solver to the problem of trusting the other prover. However, if the other prover is chosen carefully, specifically if the choice is made to use a prover that has a small set of simple core proof rules, then the result is a system in which the SMT solver can use complex proof rules, while at the same time the set of rules that must be trusted is small and simple.

In this paper, we describe our experience with this paradigm. The SMT solver is CVC3 [4], and the proof-checker is HOL Light [5]. To motivate and test the system, we applied it to benchmarks from the SMT-LIB library [3]. These benchmarks are used as points of comparison in many papers as well as in the annual SMT-COMP competition. Every benchmark in SMT-LIB contains a status field indicating whether it is satisfiable, unsatisfiable, or unknown. While benchmark providers and SMT-LIB maintainers do their best to ensure that the status fields are correct, occasionally benchmarks are incorrectly labeled resulting in confusion or controversy.

Our eventual goal is to *certify* as many unsatisfiable benchmarks as possible by producing and checking their proofs. Here, we report on our initial progress towards this goal. Ultimately, satisfiable benchmarks could (and should) also be certified by producing and checking models, but that is beyond the scope of this effort.

The paper is organized as follows. Section 2 gives a brief introduction to CVC3 and its proof system. Section 3 describes the theorem prover HOL Light. Section 4 discusses the translation procedure and several obstacles that had to be overcome in order to make it work. Section 5 discusses our experience running the system on the SMT-LIB benchmarks. Section 6 discusses related work, and Section 7 concludes.

2 CVC3

CVC3 is the latest in a series of SMT solvers (CVC, CVC Lite, CVC3). It aims to be both a platform for SMT research as well as a robust tool for use in verification applications. CVC3 is open-source, is maintained by a number of contributing developers, and enjoys a large and active user community. In order to achieve competitive performance on large benchmarks, CVC3 employs a number of optimization strategies which complicate the code. For instance, CVC3 implements its own memory manager, has reference counting schemes for expressions and theorems, and uses sophisticated data structures for backtracking. At the time of this paper, the code base consists of nearly 100,000 lines of intricate C++ code.

Because applications of theorem provers like CVC3 need to be able to rely on correct results, it is of the utmost importance that the complexity of CVC3 not compromise its correctness. In particular, a theorem prover that is unsound (i.e. reports that a theorem is unsatisfiable when it is actually satisfiable) could lead to missed bugs in critical applications.

One of the primary goals with the CVC family of systems has been to have high confidence in their soundness. The first system, CVC, pioneered the use of proofs within a state-of-the-art SMT solver [9]. The current system, CVC3, builds upon a proof infrastructure developed for CVC Lite [2]. Here, we give a brief overview of CVC3's proof system.

2.1 Proofs

A *proof* is a tree in which each node is labeled with a formula. The formulas at the leaves of the tree are called *assumptions* and the formula at the root is called the *conclusion*. Assumptions may be designated as *open* or *closed*.

A *sequent* is a pair $\Gamma \vdash \phi$, where Γ is a set of formulas and ϕ is a formula. Since we are often interested only in the assumptions and the conclusion, the sequent $\Gamma \vdash \phi$ is used to represent any proof whose open assumptions are Γ and whose conclusion is ϕ.

A *proof rule* or *inference rule* is a function which takes one or more proofs (called *premises*) and returns a new proof (the *consequent*) whose root node has each of the input proofs as its children. A proof rule specifies what formula should label the new root node and may also change the designation of one or more assumptions from open to closed.

Proof rules depend only on the assumptions and conclusions of their premises and can thus be described using sequents. We denote a proof rule as follows:

$$\frac{P_1 \quad \cdots \quad P_n}{C}$$

where the P_i's are sequents representing the premises and C is a sequent representing the new proof tree. The proof rule takes any set of proofs which match the P_i's and returns a new proof whose root is labeled by the right-hand side of C. If an assumption appears in some P_i but not in C, then that assumption is closed in the proof tree constructed by the proof rule. If there are no premises, the rule is called an *axiom*.

A sequent $\Gamma \vdash \phi$ is *valid* if the conjunction of the assumptions in Γ implies ϕ. A proof rule is *sound* if the validity of all its premises implies the validity of the conclusion. It is not hard to see that if all the proof rules are sound, then any sequent representing a proof constructed using those proof rules is valid.

2.2 Proof Rules

The most basic rule is the assumption axiom. This rule, together with a few other simple rules, are shown below.

$$\frac{}{\phi \vdash \phi}\ \text{assume}$$

$$\frac{\Gamma_1 \vdash \phi \leftrightarrow \psi \quad \Gamma_2 \vdash \psi \leftrightarrow \theta}{\Gamma_1 \cup \Gamma_2 \vdash \phi \leftrightarrow \theta}\ \text{iffTrans}$$

$$\frac{\Gamma_0 \vdash \alpha_0 \quad \Gamma_1 \vdash \alpha_1 \quad \ldots \quad \Gamma_n \vdash \alpha_n}{\Gamma_0 \cup \Gamma_1, \ldots, \Gamma_n \vdash \phi \leftrightarrow \phi'}\ \text{simplify}$$

Some proof rules (like the middle one above), have results that are completely determined by the premises. Others (like the other two) require additional parameters. For instance, `assume` has no premises and takes ϕ as a parameter, producing the sequent $\phi \vdash \phi$. Similarly, `simplify` takes a set of premises $\Delta = \{\Gamma_i \vdash \alpha_i \mid i \in \{0 \ldots n\}\}$ and the formula ϕ to be simplified as a parameter. It returns a sequent for $\phi \leftrightarrow \phi'$ where ϕ' is obtained by replacing all instances of the literals in Δ by *true* (and their negations by *false*) and applying simple Boolean rewrites to the result.

At the time this paper was written, there were 298 proof rules in CVC3. They include basic first-order rules, rules for propositional logic, and a vari-

ety of rules for theory-specific reasoning. They range from extremely simple to very complex.

2.3 Implementation

In CVC3, one of the basic classes is the `Theorem` class. Each instance of this class represents a proof and contains the sequent, i.e. the assumptions and conclusion for this proof. These `Theorem` objects exist even in the high-performance non-proof-producing version of the code. In fact, the assumption lists are critical for producing conflict clauses (see [2]). If proof-production is enabled, then each `Theorem` object in addition contains the actual proof tree represented as a directed acyclic graph (i.e. identical subtrees are shared). Each proof rule is implemented as a function which takes 0 or more `Theorem` objects (the premises) as well as any necessary parameters as input and produces a new `Theorem` (the consequent) as output. These functions exist in specially designated *trusted* code modules. A compile-time check ensures that only trusted modules can create new `Theorem` objects. In addition, each proof rule function checks that its premises are of the right form. These features help ensure that soundness bugs can only be the result of problems in the trusted code modules.

Implementing proof production has been valuable in helping shape and understand the design of the system. More importantly, it has caught and prevented bugs in CVC3. Recently, some changes to the arithmetic module uncovered a soundness bug (a previously known satisfiable benchmark was reported unsatisfiable). We ran our translator and found one step of the proof that could not be validated. A careful examination of the proof rule in question showed that the proof rule itself was not sound. This bug had persisted in CVC3 for years and would have been extremely difficult to find without the proof system.

3 HOL Light

HOL Light is a general purpose interactive theorem prover based on higher order logic. Like other HOL-like systems, HOL Light is capable of formalizing most of useful mathematics. In particular, it is capable of formalizing the theories and reasoning used in SMT solvers.

HOL Light is built on top of a very small trusted logical core. The logical core implements a proof system consisting of ten inference rules, mostly about equality, three axioms, and two principles of conservative definitional extension. It is implemented using only 430 lines of Ocaml code. Except for equality, all other logical symbols are defined, even the propositional connectors like "$\wedge$" and "$\vee$". HOL Light's definitional extension mechanism guarantees that each definition is sound and that any theorems proved are valid as long as the logical core is valid. In addition to being small, the

majority of the trusted core of HOL Light has itself been verified formally [6]. As John Harrison, the author of HOL Light, has stated, "it sets a very exacting standard of correctness"[1].

HOL Light is programmable and can easily be extended. Derived proof rules and decision procedures can be implemented as Ocaml functions. Many such derived functions exist as part of HOL Light already. For example, the function *REAL_ARITH* is a decision procedure for proving basic facts about arithmetic. HOL Light also includes decision procedures for propositional and first-order reasoning. These tools can be leveraged for our proof-checking purposes.

4 Proof Translation

When CVC3 is presented with a verification task in SMT-LIB format, it may respond with "satisfiable", "unsatisfiable", or "unknown" (or it may timeout or run out of memory). When the result is "unsatisfiable", CVC3 can produce a proof using its proof system. This proof is used as input to a translation program written on top of HOL Light. The goal of the translator is to read the CVC3 proof and reproduce the same reasoning steps in HOL Light. In order to do this, the translator must be able to translate both the formulas and the proof rules. In this section we discuss how this is done with emphasis on specific challenges that had to be overcome.

4.1 Translation of formulas

CVC3 uses the language of many-sorted first-order logic, while HOL Light is based on higher order logic. Because the theories used in CVC3 can be defined (or are already defined) in HOL Light, it is fairly straightforward to translate formulas of CVC3 into formulas of HOL Light. There are, however, a few idiosyncrasies of the SMT-LIB format that are a bit challenging for HOL Light.

For example, SMT-LIB supports a built-in predicate of variable arity called `distinct`. $\texttt{distinct}(x_1, x_2, ..., x_n)$ means $\forall\, i\, j : [1 \ldots n].\ (i \neq j \rightarrow x_i \neq x_j)$. Because predicates in HOL Light must have a fixed arity, we model this predicate by defining a set of parametrized predicates $\texttt{distinct}_n$, where n is the arity. These predicates are defined only when needed by the translator.

The translation of variables and constants of real and integer types is a bit tricky. CVC3 allows integers to be used as arguments to real operators. This is not allowed in HOL Light. Thus, during translation, if integers and reals appear in the same formula, the integers are lifted into reals.

4.2 Translation of proof rules

For each proof rule in CVC3, we write an Ocaml function whose purpose is to get HOL Light to prove the conclusion given HOL Light theorems for the premises. A naive approach is just to call built-in HOL Light functions and hope they will succeed. For instance, we could call the built-in HOL Light function `REAL_ARITH` to do reasoning about arithmetic. This approach sometimes works for simple rules and formulas, but is too slow to use in general.

A much better method is to prove generalized versions of each proof rule in HOL Light ahead of time and then just instantiate these theorems. For example, consider the following CVC3 proof rule (where x and y are parameters that must be integers):

$$\frac{}{\vdash x < y \leftrightarrow x \leq y - 1}\ \text{lessThanEqRhs}$$

The corresponding theorem in HOL Light is `!x y. x:int < y <=> x <= y + (-- &1)`. When translating the proof rule `lessThanEqRhs`, we first translate the terms that are used as parameters and then use them to instantiate this theorem. Instantiation of existing theorems is highly efficient in HOL Light and can be used whenever a proof rule corresponds directly with a HOL Light theorem.

Sometimes a proof rule cannot be represented by a single theorem. For example, CVC3's `or_distributivity` rule generates a theorem $(A \wedge B_1) \vee (A \wedge B_2) \vee ... \vee (A \wedge B_n) \leftrightarrow A \wedge (B_1 \vee B_1 \vee ... \vee B_n)$. For such proof rules, We can create a customized theorem on the fly and then instantiate it. In this case, we prove a HOL Light theorem in which A and each B_i are replaced by universal propositional variables. Even for such cases, instantiation of a general theorem is typically faster than proving the particular instance of the proof rule directly because the formulas appearing in the instance (i.e. those used to instantiate A and the B's) could be arbitrarily complex.

One special rule in CVC3 is the skolemization rule:

$$\frac{}{\vdash \exists x.P(x) \leftrightarrow P(c)}\ \text{skolem}$$

In this rule, c is a particular constant that is always used as the witness for $P(x)$. In classical first-order logic, this rule is actually unsound. However, we can use HOL Light's *choice operator* to translate this rule soundly. We simply translate the special constant c as $\varepsilon x.P$. Here, $\varepsilon x.P$ means some x that makes P true if there is one.

Sometimes CVC3 and HOL Light have similar proof rules. For example, `subst_op` in CVC3 and `SUB_CONV` in HOL Light are both rules for substitutions. However, the `subst_op` has a variant in which only some of the children are substituted. For this rule, a modification of the existing code for `SUB_CONV` can be used to do the translation.

Finally, some proof rules are too complicated for any of these approaches and custom translation functions must be written for them. An example is the `rewrite_and` rule. This rule flattens nested conjunctions, removes conjunctions containing *true*, and performs several other rewrites on conjunctions. We wrote a custom translation function that makes use of a proof rule in HOL Light for rewriting conjunctions.

4.3 Translation of propositional reasoning

Most modern SMT solvers use a separate SAT solver to do propositional reasoning and CVC3 is no exception. In previous versions of CVC, proofs could only be obtained by using a slower, custom-built, proof-producing SAT solver [2]. However, modern SAT solvers like zChaff and Minisat can dump a resolution proof for unsatisfiable formulas. We followed a similar approach to that taken by others (e.g. [11]) to produce a complete proof, given the resolution proof.

The rule for propositional resolution can be described as follows.

$$\frac{\Gamma_1 \vdash A \vee B \quad \Gamma_2 \vdash \neg A \vee C}{\Gamma_1 \cup \Gamma_2 \vdash B \vee C}\ \text{bool\_resolution}$$

We experimented with several methods of implementing the resolution rule in HOL Light. One possibility, if A is to be resolved, is to first reorder the disjunctions and move A and $\neg A$ to the front of their respective clauses, removing any duplicate occurrences at the same time. We can then instantiate the following theorem: $(A \vee B) \wedge (\neg A \vee C) \leftrightarrow (B \vee C)$. Unfortunately, translating a large resolution proof using this method turns out to be quite slow. Fortunately, there is a better way. As described in [10], the representation of CNF clauses can be changed into so-called *Sequent Representation*. The key idea is to represent the literals of a clause as assumptions. For instance, $\Gamma \vdash A \vee B$ is represented as $\Gamma, \neg A, \neg B \vdash False$. HOL Light uses a set to store the assumptions, so no reordering is needed, and when two assumption lists are merged, the duplicated literals are removed automatically. In the latest version of HOL Light, the assumption lists are ordered, which further speeds up propositional resolution.

A final point about translating the propositional reasoning has to do with the initial conversion to CNF before running the SAT solver. CVC3 uses a standard Tseitin-style conversion algorithm which introduces additional variables. Each of these new variables is a placeholder for some other formula. However, proving the equisatisfiability of the CNF formula with all of these new variables is an unnecessary complication. Instead, as we replay the resolution proof, we substitute the original formulas for the placeholder variables at each step. Care must be taken to distinguish for instance, the clause $A \vee B \vee C$ from the clause $p \vee C$ where p is a placeholder for $A \vee B$, but with this caveat, there is no difficulty checking the resolution proof. Notice

that, in particular, the clauses introduced by converting some internal node in the formula to CNF are all tautologies, and can thus be proved easily by HOL Light.

4.4 Final check

If the proof translation in HOL Light succeeds, the result is a theorem of HOL Light. However, it may be difficult to determine, especially for large problems, whether the theorem proved by HOL Light is in fact the same as the original problem posed to CVC3. A bug in the proof or the proof translation could result in HOL Light successfully proving the wrong theorem. To eliminate the need to trust the proof or the proof translation, we added one final check to make sure that theorem proved in HOL Light is indeed the theorem we want to prove. This is done by translating the original problem into HOL Light's language directly and comparing the result to the theorem that was proved. This reduces the code that must be trusted (besides HOL Light) to CVC3's parser and the formula translator, which are trivial compared to the rest of the system.

5 Experimental results

We tested the system on a subset of the AUFLIA benchmarks from SMT-LIB. All tests were run on AMD Opteron-based (64 bit) systems with 2GB of RAM running Linux. CVC3 was given a time limit of one minute. The translator was given a time limit of 10 minutes.

Table 1 summarizes the results. Each row shows, from left to right, the total number of unsatisfiable cases in this family, the number of cases that CVC3 successfully proved to be unsatisfiable, the total and average time spent by CVC3 for the family, the number of cases for which the proof translation succeeded, and the total and average time for the translation. As seen from the table, the amount of time spent for the proof translation in HOL Light varies significantly, from less than 10 times the amount of time spent in CVC3 to nearly 100 times the amount of time spent in CVC3. There are several cases in the `simplify2` family for which the proof translation fails. Some of these time out, and others contain proof rules not yet fully supported by the translator.

The table does not list results for the "boogie" family of benchmarks because complete results are not yet available. However, we did find something very interesting. CVC3 reported unsatisfiable for two cases in labeled as satisfiable. We were able to translate these proofs successfully, meaning that these two benchmarks were incorrectly labeled.

category		CVC3			HOL Translation		
	Cases	Proved	Total	Ave	Proved	Total	Ave
simplify	833	833	814.30	0.98	833	16249.33	19.51
simplify2	2329	2306	2408.95	1.11	2164	19153.34	8.85
Burns	14	14	0.30	0.02	14	19.37	1.38
Ricart	14	13	0.89	0.07	13	228.80	17.60
piVC	41	41	4.92	0.12	41	59.40	1.45

Table 1: Results on a selection of AUFLIA benchmarks

6 Related work

Moskal [8] proposed a rewriting system for proof checking of SMT solvers. His implementation emphasizes speed and compactness. Our system, while slower, emphasize trustworthiness. Our system ultimately provides a very strong guarantee of correctness, and essentially none of the code of the SMT solver or the proof translator need be trusted.

Our own previous work in this direction [7] described our initial efforts to combine CVC Lite and HOL Light. That work emphasized using CVC Lite as an external decision procedure for HOL Light. Here, we emphasize HOL Light's value as a proof-checker for CVC3. We also give a more detailed description of the system and give results on the SMT-LIB benchmarks.

7 Conclusion

CVC3 is an SMT solver with many diverse proof rules. The proof rules were designed to enable high-performance and to be convenient for CVC3 developers. This is at odds with the goal of having a small and simple trusted core. The problem can be alleviated by translating proofs into another prover that does have a small trusted core, in this case HOL Light. We plan to continue our efforts to certify as much of the SMT-LIB library as possible. We also plan to continue improving and tuning the translator.

8 Thanks

We would like to thank John Harrison for his help with HOL Light, and we would also like to thank Sean McLaughlin for writing the first version of the proof translator.

References

[1] `http://www.cl.cam.ac.uk/~jrh13/hol-light/`.

[2] C. Barrett and S. Berezin. A proof-producing boolean search engine. In *Proceedings of the* 1^{st} *International Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR '03)*, July 2003. Miami, Florida.

[3] C. Barrett, S. Ranise, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2008.

[4] C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany.

[5] J. Harrison. Hol light: A tutorial introduction. In M. K. Srivas and A. J. Camilleri, editors, *FMCAD*, volume 1166 of *Lecture Notes in Computer Science*, pages 265–269. Springer, 1996.

[6] J. Harrison. Towards self-verification of hol light. In U. Furbach and N. Shankar, editors, *IJCAR*, volume 4130 of *Lecture Notes in Computer Science*, pages 177–191. Springer, 2006.

[7] S. McLaughlin, C. Barrett, and Y. Ge. Cooperating theorem provers: A case study combining HOL-Light and CVC Lite. In A. Armando and A. Cimatti, editors, *Proceedings of the* 3^{rd} *Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR '05)*, volume 144(2) of *Electronic Notes in Theoretical Computer Science*, pages 43–51. Elsevier, Jan. 2006. Edinburgh, Scotland.

[8] M. Moskal. Rocket-fast proof checking for smt solvers. In K. Jesen and A. Podelski, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 486–500. Springer, 2008.

[9] A. Stump and D. Dill. Faster proof checking in the edinburgh logical framework. In A. Voronkov, editor, *Proceedings of the* 18^{th} *International Conference on Automated Deduction (CADE '02)*, volume 2392 of *Lecture Notes in Computer Science*, pages 185–222. Springer, 2002.

[10] T. Weber. Efficiently checking propositional resolution proofs in isabelle/hol. volume 212 of *CEUR Workshop Proceedings*, 2006.

[11] T. Weber. Integrating a SAT solver with an LCF-style theorem prover. In A. Armando and A. Cimatti, editors, *Proceedings of the* 3^{rd} *Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR '05)*, volume 144(2) of *Electronic Notes in Theoretical Computer Science*, pages 67–78. Elsevier, Jan. 2006. Edinburgh, Scotland.

Towards an SMT Proof Format

Aaron Stump and Duckki Oe
Computer Science and Engineering
Washington University in St. Louis
St. Louis, Missouri, USA

Abstract

The Edinburgh Logical Framework (LF) extended to support side condition code (LFSC) is advocated as a foundation for a proof format for SMT. The flexibility of the framework is demonstrated by example encoded inference rules, notably propositional resolution. Preliminary empirical results obtained with a SAT solver producing proofs in LFSC format are presented.

1 Introduction

Given the complexity of modern SMT solvers, practical methods for verifying either the solvers or their results are desirable. A well-known method for verifying reports of unsatisfiability is for solvers to produce a proof refuting the formula. Just as was done by the SMT-LIB initiative for SMT input formulas, it is highly desirable to develop a common format for such proofs, which different solvers could target. Since SMT solvers employ a variety of reasoning methods, it is at the least premature to standardize around a particular set of axioms and inference rules. Instead, this paper proposes to standardize around a logical framework, in which proof systems corresponding to these different reasoning methods can be encoded. This framework is called LF with Side Conditions (LFSC), and is an extension of the Edinburgh Logical Framework (LF), which has been used successfully for representing proofs in applications like proof-carrying code [3, 1]. LF allows proof systems to be encoded succinctly in a type theory with special support for variable-binding constructs, which occur frequently in both formula and proof syntaxes. LFSC extends LF with better support for rules with computational side conditions. Encoded inference rules may require certain side conditions to succeed for every application of the rule, where the side conditions are described using a simple functional programming language. This paper introduces the syntax and informal semantics of LFSC (Section 2), and shows how LFSC can be used to encode resolution inferences (Section 3). Empirical results using a solver called CLSAT are presented (Section 4).

$(\mathit{declare}\ c\ T) \quad || \quad (\mathit{define}\ c\ t) \quad || \quad (\mathit{opaque}\ c\ t) \quad || \quad (\mathit{check}\ t)\ ||$
$(\mathit{program}\ c\ ((x_1\ t_1)\ \cdots\ (x_{n+1}\ t_{n+1}))\ t\ C)$

Figure 1: Top-level commands.

2 LF with Side Conditions

This section presents the syntax and gives an informal description of the typing and operational semantics for LF with Side Conditions (LFSC). Commands are described first. Then terms (t), types (T), and side condition code (C).

2.1 Command Syntax

A proof is given as a list of commands, listed in Figure 1 and described next:

Declarations. Constants c may be declared (with *declare*) to have type T, or may be defined (with *define*) to equal t. They may also be given an opaque definition (with *opaque*), in which case subsequent type checking will use only the fact that c has type T computed for t, and not the fact that c equals t. Declared constants are used in the LF encoding methodology to represent primitive symbols of an object-language. For SMT, these include theory functions and predicate symbols. They are also used to represent axioms and inference rules. Opaque definitions ($\mathit{opaque}\ c\ t$) are introduced in LFSC to allow the memory required for the defining term t to be reclaimed after checking its type. Ordinary definitions (introduced with *define*) must retain the defining term, in case the subsequent proof depends on the fact that c equals t.

Checks. We can check a term t by computing a type for it.

Programs. Singly recursive programs may be defined at the top-level of the input using *program* commands (adding support for mutual recursion would not be difficult). The recursive program is called c; its input variables are $x_1, \ldots, x_{n+1}$, with types $t_1, \ldots, t_{n+1}$; its return type is t, and its body is C.

One simple usage scenario for these commands is the following. Following standard LF terminology, a collection of declarations and definitions is called a *signature*. The SMT-LIB organization may provide a signature specifying SMT input syntax for various logics. It is also desirable to provide an example signature for axioms and inference rules for that syntax, perhaps based on the one in progress by the authors. SMT solver implementors may use this example signature, or write their own to specify their axioms and inference rules. Untrusted proofs are then not to use declarations, since these may non-conservatively extend the logic; but only definitions and *check* commands.

2.2 Term and Type Syntax

Figure 2 gives the syntax for LFSC terms t and types T. In multi-arity notation such as $(t_1\ \cdots\ t_{n+2})$, the number n is a natural number (including possibly

$$
\begin{array}{rcl}
t & ::= & x \mid\mid c \mid\mid \_ \mid\mid N \mid\mid (t_1 \ \cdots \ t_{n+2}) \mid\mid (\backslash \ x \ t) \mid\mid (\$ \ x \ T \ t) \mid\mid (: \ T \ t) \\
T & ::= & c \mid\mid (T \ t_1 \ \cdots \ t_{n+1}) \mid\mid (! \ x \ r \ T) \\
r & ::= & (\hat{} \ C \ t) \mid\mid T
\end{array}
$$

Figure 2: Syntax for terms (t) and types (T).

0); so for this example, at least two terms must be present in the list of terms $t_1, \ldots, t_{n+2}$. We briefly describe the various constructs, and how they are used in representing theories.

Variables and constants. We write x for variables, c for constants, and N for unbounded integer literals (adding support also for rational literals would be straightforward). The following special constants c are assumed present: *type*, which is the type for types; and *mpz*, the type for integers.

Applications. Term-level applications ($t_1 \ \cdots \ t_{n+2}$) consist of a functional term t_1 and its arguments $t_2 \ \cdots \ t_{n+2}$. Applications are used to represent applications of SMT function and predicate symbols to arguments. They are also used to represent applications of logical connectives to formulas. Finally, they are also used to represent applications of inference rules and axioms to arguments for their quantified variables and hypotheses.

Holes. Applications may use holes _, whose value is to be filled in from the types of subsequent arguments.

Indexed types. Type-level applications ($T \ t_1 \ \cdots \ t_{n+1}$) are used for indexed types. In a standard functional language, we can easily define a datatype `Pf` for proofs. But type checking cannot enforce proof checking, because standard type systems do not have a way to express that a proof is a proof of a given formula. Thus, we could apply an encoded inference rule like modus ponens to any two terms of type `Pf`, regardless of whether or not the first is a proof of $\phi \rightarrow \psi$ and the second is a proof of ϕ. With indexed types, the type system of LF (and LFSC) is able to track the formula proved by a particular proof, as an index to the type. So an encoded proof of ϕ would have type "`(Pf F)`", assuming `F` is the encoding of formula ϕ. Using indexed types, axioms and inference rules can be encoded as term constructors in such a way that they cannot be applied in cases where the object-language inference would be incorrect. We can, for example, declare (encoded) modus ponens to be a term constructor accepting one argument of type "`(Pf (imp F1 F2))`", and another argument of type "`(Pf F1)`", and constructing a resulting term of type "`(Pf F2)`".

Lambda abstractions. The expressions ($\backslash \ x \ t$) are conventionally written in mathematical notation $\lambda x.\ t$. They are anonymous functions accepting arguments for input variable x, and returning t. In the LF encoding methodology, lambda abstractions are used to encode object-language binding constructs.

$$
\begin{array}{rcl}
C & ::= & x \mid\mid c \mid\mid N \mid\mid (\odot\ C_1\ \cdots\ C_{n+1}) \mid\mid (c\ C_1\ \cdots\ C_{n+1}) \\
& & \mid\mid (\mathit{match}\ C\ (P_1\ C_1)\ \cdots\ (P_{n+1}\ C_{n+1})) \mid\mid (\mathit{do}\ C_1\ \cdots\ C_{n+1}) \\
& & \mid\mid (\mathit{let}\ x\ C_1\ C_2) \mid\mid (\mathit{markvar}\ C) \mid\mid (\mathit{ifmarked}\ C_1\ C_2\ C_3) \mid\mid (\mathit{fail}\ T) \\
\\
P & ::= & (c\ x_1\ \cdots\ x_{n+1}) \mid\mid c
\end{array}
$$

Figure 3: Syntax for code (C) and patterns (P).

The variable bound by an object language binder is represented by a λ-bound variable in LF. It is customary with LF to omit the type for the variable x, because the LF type checking algorithm will fill it in from the context surrounding the lambda abstraction. It turns out that for representing large proofs, it can be necessary to allow the type for the bound variable to be specified in the λ-abstraction. In this case, we write ($\$\ x\ T\ t$) for ($\lambda\, x : T.\ t$).

Pi abstractions. The expressions ($!\ x\ t_1\ t_2$) are conventionally written in mathematical notation $\Pi\, x : t_1.\ t_2$. These are the types for functions accepting inputs x of type t_1 and producing outputs of type t_2, where t_2 may contain x free. Such types are called *dependent function types*, because the output type can depend on the input argument. A programming example may help provide intuition for such types. Imagine a function that accepts a natural number n and returns an array of *int*s of length n. Such a function might be given dependent function type $\Pi\, x : \mathit{nat}.\ \mathit{int}[n]$ (writing $\mathit{int}[n]$ for the type of arrays of *int*s of length n). Such dependencies are used heavily when giving the types for encoded inference rules, since we wish to capture dependencies such as described above between the indices of the types of subproofs of modus ponens. As its crucial addition to standard LF, LFSC allows the domain type of an abstraction to be of the form ($\hat{}\ C\ t$). The intuitive meaning of this expression is that running code C should produce result t. Note that t may be a hole, in which case it will be filled in with the output produced by running C.

Ascriptions. Expressions ($:\ T\ t$) state that T is the type for term t. Type checking an ascription requires verifying that this is indeed the case. Ascriptions are needed just in giving definitions.

2.3 Code Syntax

Figure 3 gives the syntax for code (C). We write $\odot$ for arithmetic operations on unbounded integers. Code constructs are used for writing strongly typed first-order, monomorphic functional programs. First-order means that unlike well-known functional languages such as HASKELL and OCAML, functions may not be passed as arguments to programs or produced as results. Monomorphic means that, again unlike those languages, programs cannot be defined polymorphically, to operate uniformly on all types of data. The programs are

mostly without mutable state, although there is a feature for marking LF variables, demonstrated below. The resulting language supports programs where inductively defined data may be recursively decomposed (using `match`) and constructed. Additionally, code can fail, either explicitly using *fail*, or by failing to match a piece of inductively defined data against any of the patterns in a match expression.

Application. Expressions $(c\ C_1\ \cdots\ C_{n+1})$ are either of term constants or program constants to arguments. In the former case, the application is constructing a new piece of inductive data. In the latter, it is invoking a program.

Match. Expressions $(match\ C\ (P_1\ C_1)\ \cdots\ (P_{n+1}\ C_{n+1}))$ evaluate C to a piece of inductively defined data, and then seek to match that piece of data against one of the given simple patterns $P_1, \ldots, P_{n+1}$. Successfully matching against a pattern P_i binds the appropriate subdata to the variables in the pattern. The body C_i of the match is then evaluated and its result returned for the result of the match expression.

Do. Expressions $(do\ C_1\ \cdots\ C_{n+1})$ evaluate each of $C_1, \ldots, C_{n+1}$ in turn, and return the value of the last. This is useful for checking that several conditions in a row do not fail (it is not related to `do` in HASKELL).

Let. Expressions $(let\ x\ C_1\ C_2)$ are as standard in functional languages. The value (if any) of C_1 is substituted for x before evaluating C_2.

Markvar. The code $(markvar\ C)$ first evaluates C. If the result is an LF variable, then this toggles a mark on that variable, and then returns the variable. These marks are useful in implementing the important example of resolution inferences below (Section 3).

Ifmarked. The code $(ifmarked\ C_1\ C_2\ C_3)$ evaluates C_1. If the result is not a variable, evaluation fails. Otherwise, if the result is marked, we evaluate C_2; otherwise, we evaluate C_3.

Fail. We have $(fail\ T)$ for explicitly indicating failure. The fail term is treated as having the given type T.

3 Encoding Resolution Proofs

This section presents an important example of encoding a proof system in LFSC, namely propositional resolution. Resolution is used during clause learning in state-of-the-art SAT and SMT solvers [5]. A proof format proposed in 2005 by van Gelder for SAT solvers is based directly on resolution. Encoding a resolution proof system in LFSC is thus a critical step in encoding proof systems for the more general logics of SMT.

3.1 Encoding Clauses

The first step in encoding any proof system in LFSC (or pure LF) is to encode the formulas of the logic. Once these are encoded, we can consider how to encode proofs. In the case of propositional resolution, the formulas of the logic are propositional clauses. Figure 4 lists LFSC declarations for these. We first

```
(declare var type)

(declare lit type)
(declare pos (! x var lit))
(declare neg (! x var lit))

(declare clause type)
(declare cln clause)
(declare clc (! x lit (! c clause clause)))
```

Figure 4: Clauses

declare an LFSC type `var`, for propositional variables. We do not give any term constructors for variables, for reasons which will be given shortly. Next, we declare a type `lit` for propositional literals, with two constructors, `pos` and `neg`. We will use these for positive and negative occurrences, respectively, of a variable in a clause. Note that the type given for both `pos` and `neg`, namely

```
(! x var lit)
```

says that `pos` and `neg` are functions taking input `x`, which is a variable, and producing a literal. Finally, the type `clause` of clauses is declared with constructors `cln` for the empty clause, and `clc` for cons'ing a literal (`x`) onto the front of a clause (`c`). Assuming that P and Q are propositional variables, then a clause like $P \vee \neg Q$ is encoded with these declarations as

```
(clc (pos P) (clc (neg Q) cln))
```

3.2 Encoding Resolution Inferences

Figure 5 gives three declarations for encoding binary resolution with factoring. More complex forms of resolution such as hyperresolution should also be encodable, but this is future work. First, we declare an indexed type `holds`, with the intention that "(`holds C`)" is the type for proofs that clause `C` holds. We defer consideration of `resolve`. The declaration of `R`, which encodes the resolution inference rule, states that `R` takes clauses `c1`, `c2`, and `c3` as inputs; then a proof that clause `c1` holds (this is the input argument `u1` of type "`(holds c1)`"), and a proof that clause `c2` holds; and a variable `v`. If running the code "`(resolve c1 c2 v)`" produces clause `c3`, then `R` constructs a term of type "`(holds c3)`", representing a resolution inference deriving `c3`. Since the values of the inputs `c1`, `c2`, and `c3` to `R` can all be filled in from subsequent arguments (in particular, `u1` to fill in `c1`, `u2` to fill in `c2`, and `r` to fill in `c3`), we can use holes "_" for these inputs whenever `R` is used. This is demonstrated in Section 3.3 below.

The side condition to check for a resolution inference is as follows. If we are resolving clauses `c1` and `c2` on variable `v`, `v` must occur positively in `c1` and

```
(declare holds (! c clause type))

(program resolve ((c1 clause) (c2 clause) (v var)) clause
  (let pl (pos v)
  (let nl (neg v)
  (do (in pl c1)
      (in nl c2)
      (let d (append (remove pl c1) (remove nl c2))
         (dropdups d))))))

(declare R (! c1 clause (! c2 clause (! c3 clause
           (! u1 (holds c1)
           (! u2 (holds c2)
           (! v var
           (! r (^ (resolve c1 c2 v) c3)
            (holds c3)))))))))
```

Figure 5: Propositional resolution in LFSC

negatively in `c2`. All positive occurrences are removed from `c1` and all negative ones from `c2`. The resulting clauses are concatenated to produce the resolvent `c3`. Additionally, it is desirable to drop duplicate literals from the resolvent.

LFSC code to compute a resolvent from `c1`, `c2`, and `v` is given in the program `resolve` of Figure 5. This program relies on several helper programs, mostly relegated to the Appendix. Reading through the body of `resolve`, we can see that it first `let`-defines `pl` to be the positive literal for variable `v`, and `nl` to be the negative literal. It then checks that this positive literal is in `c1`, and the negative one in `c2`. It then `let`-defines `d` to be the result of appending the results of removing the positive literal from `c1` and the negative literal from `c2`. Finally, we return the result of dropping duplicates from `d`.

3.3 An Example Resolution Proof

For a very simple example, suppose our clause database consists of the following clauses: $\neg V_1 \vee V_2$, then $\neg V_2 \vee V_3$, then $\neg V_3 \vee \neg V_2$, and $V_1 \vee V_2$. A resolution derivation of the empty clause from these clauses is given in Figure 6. The encoding in LFSC of this proof is given in Figure 7. We can see in the last line of Figure 7 three applications of `R`, corresponding to the three resolution inferences in Figure 6. As mentioned above, the clauses being resolved and the resolvent (the first three arguments to `R`) do not need to be mentioned when `R` is applied, because those clauses can all be filled in from subsequent arguments. That is why each use of `R` begins with three holes. The nested resolutions in Figure 6 are mirrored by the nested applications of `R` in Figure 7. Where Figure 6 lists the n'th clause (in the order the clauses were listed above) from our clause database, at the leaves of the proof tree; in those places Figure 7 lists

$$\dfrac{\dfrac{V_1 \vee V_2 \quad \neg V_1 \vee V_2}{V_2} \quad \dfrac{\neg V_2 \vee V_3 \quad \neg V_3 \vee \neg V_2}{\neg V_2}}{\mathit{empty}}$$

Figure 6: An example refutation

```
(check ($ v1 var ($ v2 var ($ v3 var
       ($ x0 (holds (clc (neg v1) (clc (pos v2) cln)))
       ($ x1 (holds (clc (neg v2) (clc (pos v3) cln)))
       ($ x2 (holds (clc (neg v3) (clc (neg v2) cln)))
       ($ x3 (holds (clc (pos v1) (clc (pos v2) cln)))
       (R _ _ _ (R _ _ _ x3 x0 v1) (R _ _ _ x1 x2 v3) v2)))))))
```

Figure 7: LFSC encoding of the example refutation

`xn` (e.g. `x3` and `x0` in the leftmost innermost resolution).

Let us now consider the LFSC command in Figure 7 more carefully. This is a `check` command. The proof begins by λ-abstracting (with \$) all the propositional variable and assumptions that all the initial clauses hold. The use of λ-abstraction here comes from standard LF encoding methodology, where one seeks to represent object-language variables via LF variables. This confers one of the main advantages of using LF, namely that the encoding need not explicitly describe safe renaming of or substitution for bound variables. These features are provided by LF directly for λ-bound variables, and hence are inherited for free by any encoding that represents object-language variables by LF variables. The main benefit of using LF variables for propositional variables is that we can efficiently test equality of variables in LFSC code using variable marking. This is necessary when computing the resolvent: for example, when testing whether or not the negative literal for variable `v` occurs in the second clause being resolved (see the Appendix).

4 Preliminary Empirical Results

CLSAT is a SMT solver currently supporting the QF_IDL logic. It can solve SAT benchmarks in DIMACS format and SMT benchmarks in SMT-LIB format, generating resolution proofs for unsatisfiable SAT benchmarks. Proofs of SMT benchmarks are not yet supported. The use of resolution in deriving conflict clauses from conflicting clauses is well known [5]. Learned clauses are recorded as lemmas to keep the proof size from exploding (see also Section A). A refutation of the formula is then just a sequence of lemmas, culminating in the empty clause.

The benchmarks used are from SAT Race 2008 Test Set 1, which represent modern SAT benchmarks. Since these are quite large and difficult to solve, 10 easier ones out of the 31 unsatisfiable benchmarks in the set were selected (see

benchmark	pf (s)	overhd	sz (MB)	num R	check (s)	tot overhd
E-sr06-par1	4.56	196.10%	35	14316	14.75	11.54
E-sr06-tc6b	0.96	152.63%	8.4	8708	11.68	32.26
M-c10ni_s	6.62	34.01%	43	4578	10.90	2.55
M-c6nid_s	15.58	12.82%	33	72930	48.35	3.63
M-f6b	20.76	29.51%	30	1018638	3237.22	202.24
M-f6n	16.59	35.76%	26	847567	2848.03	233.42
M-g6bid	20.05	28.61%	27	797530	1165.57	75.05
M-g7n	16.12	43.03%	28	1006820	1707.43	151.93
V-eng-uns-1.0-04	25.04	29.27%	41	1692714	5913.22	305.57
V-sss-1.0-cl	4.18	46.15%	9.8	416200	553.30	193.92

Figure 8: Proof sizes and proof checking times

also Section D). Checking is carried out using a prototype LFSC checker [4]. The checker does not yet compile side condition code. Profiling reveals that around 90% of checking time is currently devoted to interpreting side condition code. This at least partially explains the much greater times required for checking proofs over producing them. The checker implements an optimization called incremental checking, where parsing and proof checking are interleaved. Abstract syntax trees for subterms of proofs are not constructed in memory at all, unless they are used in the theorem proved by a subproof.

The proofs of all derived lemmas are emitted by the solver. Excluding unnecessary lemmas is expected often to result in smaller proofs, but this requires storing all proofs until the end of the run. Incremental checking alone is currently not fast enough to keep up with the solver's proof production. But if this can be achieved, then the checker can consume proofs of unnecessary lemmas as they are produced. This would allow the solver to avoid storing them during the attempted refutation, and hence would be preferable, at least for proof checking purposes, to emitting just the needed lemmas.

Figure 8 shows results related to proof production and checking. The "pf" column gives the time to solve the benchmark and produce a proof (including time to write the proof to disk). The "overhd" column gives the percentage running time overhead incurred for proof production. The "sz" column gives the size of the generated proofs in megabytes. The "num R" column gives the number of resolutions. The "check" column gives proof checking time in seconds. The "totl overhd" column gives the ratio of proof production and checking time, to time needed to solve the benchmark without producing a proof. All experiments were performed on an Intel Core 2 Duo 2GHz, 4MB L2 Cache, 2GB memory, running MacOS 10.5.2.

5 Related Work and Conclusion

A recent paper by M. Moskal gives another approach to flexible proof checking, currently achieving much faster proof checking than reported here [2]. Moskal's approach uses term rewriting with a fixed (non-standard) reduction strategy to

rewrite proofs to the theorems, if any, they prove. This formalism combines symbolic and functional programming, the former because the reduction strategy includes reduction beneath λ-abstractions. This will prove a hinderance to compilation. In contrast, LFSC separates a standard, naturally compilable programming language from more declarative aspects of proofs (particularly those involving bound variables). A further difference is that Moskal's formalism is untyped, while LFSC's enjoys the well-known benefits of strong typing. Finally, Moskal's formalism includes ad hoc features to support sound skolemization. While conversion to CNF has not yet been implemented in CLSAT, LF's direct support for higher-order abstract syntax enables introduction of new symbols without additional ad hoc features.

This paper has taken important first steps towards a meta-logical approach to SMT proofs, based on LF with Side Conditions (LFSC). The important case of propositional resolution has been evaluated with the CLSAT SAT solver, which produces LFSC proofs for unsatisfiable benchmarks. This work's achievement is in supporting large propositional resolution proofs with a general meta-logic. The use of a meta-logic paves the way for flexibly supporting theory inferences and CNF conversion. Future work includes proof production for full SMT reasoning in CLSAT, and compilation of side condition code for faster LFSC proof checking.

Acknowledgments. This work was partially supported by NSF award 0551697. Thanks are due to the SMT 2008 reviewers for helpful comments that improved the paper. Thanks also to Clark Barrett, Leonardo de Moura, Pascal Fontaine, and Cesare Tinelli for helpful discussions on LF-based proof checking for SMT.

References

[1] R. Harper, F. Honsell, and G. Plotkin. A Framework for Defining Logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[2] M. Moskal. Rocket-Fast Proof Checking for SMT Solvers. In C. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

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

[4] A. Stump. Proof Checking Technology for Satisfiability Modulo Theories. In A. Abel and C. Urban, editors, *Logical Frameworks and Meta-Languages: Theory and Practice*, 2008.

[5] L. Zhang and S. Malik. The Quest for Efficient Boolean Satisfiability Solvers. In *Proceedings of 8th International Conference on Computer Aided Deduction (CADE 2002)*, 2002.

A Propositional Lemmas

To keep proofs as compact as possible, it is critical for encoded resolution proofs to be able to introduce lemmas for learned clauses. This is done with `satlem`, defined in Figure 9. Note the use of an ascription in giving this definition. If `P1` proves that clause `c1` holds, and if `P2` is a proof of clause `c2` from a hypothesis named x that `c1` holds, then "`(satlem _ _ P1 ( x P2))`" derives `c2` without hypotheses. In practice, the name of the hypothesis x can be determined by the clause number introduced by the SAT solver for the new clause.

B Helper Code for Resolution

The helper code called by the side condition program `resolve` of the encoded resolution rule `R` is given in Figures 10 and 11. We can note the frequent uses of `match`, for decomposing or testing the form of data. The program `eqvar` of Figure 10 uses variable marking to test for equality of LF variables. The code assumes a datatype of booleans `tt` and `ff`. It marks the first variable, and then tests if the second variable is marked. Assuming all variables are unmarked except during operations such as this, the second variable will be marked iff it happens to be the first variable. The mark is then cleared (recall that `markvar` toggles marks), and the appropriate boolean result returned. Marks are also used by `dropdups` to drop duplicate literals from the resolvent.

C Encoding Theory Reasoning

For SMT, it is, of course, necessary to add theory reasoning to our encoded propositional resolution calculus. The authors have not yet implemented this in CLSAT, and so these ideas must be considered preliminary. Let us distinguish sorted terms and formulas (a more unified view, as under discussion for the next revision of the SMT input syntax, should also be possible to support). If we wish, we can easily embed SMT type checking into LF type checking by using indexed types "`(term s)`" as the LF type for encodings of SMT terms of (encoded) sort `s`. Here we give an example theory inference rule, from Integer Difference Logic. This rule says that if $\neg\ x - y\ \leq\ c$ holds, then so does $y - x \leq d$, where c is an integer constant and d is the integer predecessor of c. The encoding of this inference, given in Figure 12, uses side condition code to compute the integer predecessor of c. The operations `mpz_add` and `mpz_neg` are integer operations implemented by the prototype LFSC checker. Using this rule, if `P` proves (the encoding of) $x - y \leq c$, then `(not<=<= _ _ _ _ P)` proves $y - x \leq d$, where $d = -c - 1$.

```
(define satlem (: (! c1 clause
                  (! c2 clause
                  (! u1 (holds c1)
                  (! u2 (! x (holds c1) (holds c2))
                     (holds c2)))))
               (\ c1 (\ c2 (\ u1 (\ u2 (u2 u1)))))))
```

Figure 9: Definition for propositional lemmas

```
(program eqvar ((v1 var) (v2 var)) bool
    (do (markvar v1)
        (let s (ifmarked v2 tt ff)
           (do (markvar v1) s))))

(program litvar ((l lit)) var
  (match l ((pos x) x) ((neg x) x)))

(program eqlit ((l1 lit) (l2 lit)) bool
  (match l1 ((pos v1) (match l2 ((pos v2) (eqvar v1 v2))
                                ((neg v2) ff)))
            ((neg v1) (match l2 ((pos v2) ff)
                                ((neg v2) (eqvar v1 v2))))))
```

Figure 10: Variable and literal comparison

```
(declare Ok type)
(declare ok Ok)

(program in ((l lit) (c clause)) Ok
  (match c ((clc l' c') (match (eqlit l l') (tt ok) (ff (in l c'))))
           (cln (fail Ok))))

(program remove ((l lit) (c clause)) clause
  (match c (cln cln)
           ((clc l' c' )
              (let u (remove l c')
                 (match (eqlit l l') (tt u) (ff (clc l' u)))))))

(program append ((c1 clause) (c2 clause)) clause
  (match c1 (cln c2) ((clc l c1') (clc l (append c1' c2)))))

(program dropdups ((c1 clause)) clause
  (match c1 (cln cln)
           ((clc l c1')
            (let v (litvar l)
               (ifmarked v
                  (dropdups c1')
                  (do (markvar v)
                      (let r (clc l (dropdups c1'))
                         (do (markvar v) ; clear the mark
                             r))))))))
```

Figure 11: Operations on clauses

```
(declare not<=<=
  (! x (term Int) (! y (term Int) (! c mpz (! d mpz
  (! u (th_holds (not (<= (- x y) (an_int c))))
  (! r (^ (mpz_add ( mpz_neg c) (~ 1)) d)
    (th_holds (<= (- y x) (an_int d)))))))))))
```

Figure 12: Example theory rule

benchmark	size (MB)	CLSAT	MINISAT	TINISAT
E-sr06-par1	8.4	1.54	1.46	1.43
E-sr06-tc6b	1.9	0.38	0.22	0.34
M-c10ni_s	10	4.94	43.42	7.14
M-c6nid_s	7.4	13.81	162.01	93.56
M-f6b	1.7	16.03	4.02	5.41
M-f6n	1.7	12.22	4.57	6.58
M-g6bid	1.8	15.59	3.60	3.99
M-g7n	1.1	11.27	2.75	6.46
V-uns-1.0-04	1.0	19.37	5.19	5.63
V-1.0-cl	0.18	2.86	0.41	0.21

Figure 13: Comparison of CLSAT with other solvers

D Solver Performance

CLSAT has its own lemma-learning SAT solver implementing watched literals, non-chronological backtracking, and conflict clause simplification. The SAT engine of CLSAT is primarily implemented by the second author, with other parts of CLSAT written by Timothy Simpson and Terry Tidwell. Figure 13 compares the running times of CLSAT, with proof production turned off, with those of MINISAT 2.0 beta and TINISAT 0.22. This figure shows that CLSAT is not too much slower than these two well-known fast modern solvers, and hence provides a reasonable basis for studying proof production.

Reasoning about Arrays

Aaron Bradley

Electrical & Computer Engineering, CU Colorado, USA
aaron.r.bradley@colorado.edu

Abstract. Arrays are a basic data structure in imperative programs and hardware. John McCarthy formalized arrays as a first-order theory in 1962. All modern SMT (Satisfiability Modulo Theories) solvers can decide satisfiability of quantifier-free array formulae based on a decision procedure first implemented by James King in 1969. However, the quantifier-free fragment can only express information about a finite number of positions of arrays — those positions that a formula explicitly references. Expressing constraints about segments of arrays requires either augmenting the theory with extra predicates or considering formulae with at least one quantifier alternation. The former approach was taken in a series of studies in the 1980s (Suzuki, Jefferson; Jaffar; Mateti) and then again in 2001 (Stump, Barrett, Dill, Levitt); they focus on quantifier-free fragments of theories augmented with predicates such as "sorted", "partitioned", and equality between arrays.

In joint work with Zohar Manna and Henny Sipma, I took the latter approach. I will describe the "array property fragments" of theories of arrays with integer indices (standard imperative arrays) and with uninterpreted indices ("maps"). Because formulae of the fragments can have one (limited) quantifier alternation, they allow encoding the various predicates that were previously studied in isolation, as well as other properties. The array property fragment of maps is also useful for modeling sets, multisets, and hashtables. After presenting simple decision procedures based on quantifier instantiation (which have been implemented in several SMT solvers), I will show that each of several obvious and natural extensions to the fragments makes satisfiability undecidable.

The presentation is based on Chapter 2 of my PhD thesis, which is available online at http://theory.stanford.edu/~arbrad.

Lemmas on Demand for the Extensional Theory of Arrays

Robert Brummayer and Armin Biere

Institute for Formal Models and Verification
Johannes Kepler University, Linz, Austria

Abstract. Deciding satisfiability in the theory of arrays, particularly in combination with bit-vectors, is essential for software and hardware verification. We precisely describe how the lemmas on demand approach can be applied to this decision problem. In particular, we show how our new propagation based algorithm can be generalized to the extensional theory of arrays. Our implementation achieves competitive performance.

1 Introduction

Reasoning about bit-vectors and arrays is an active area of research and is successfully applied to hardware and software verification. Particularly, deciding satisfiability of first-order formulas, with respect to theories, known as Satisfiability Modulo Theories (SMT) [1–6], increasingly gains importance.

SMT approaches can be roughly divided into *eager* and *lazy* [7]. There is a special case called *lemmas on demand* [8–12]. In principle, it is an eager translation of the formula, where theory constraints are translated on demand, rather than up front. The SAT solver is used as a black box and generates concrete assignments. If an assignment violates the theory, then a lemma that rules out this assignment is added as refinement. An important aspect of this approach is that the theory solver is used as consistency checker for concrete assignments.

Previous work [8–12] lacks a precise description on how lemmas on demand can be generated for arrays, particularly, if equalities on arrays are involved. Introducing array equalities, particularly inequalities, makes the theory extensional. Prior work on arrays typically maintains congruence on arrays, but not necessarily extensionality. Our approach handles both.

Extensionality has many applications in hardware and software verification. A typical example in hardware is conformance of a pipelined machine to a sequential implementation of its instruction set architecture, in particular with memory. On the software side, extensionality makes it easy to specify that algorithms have the same memory semantics. In both cases, the algorithms respectively machines are symbolically executed on one big array, which models memory. Finally, the memories are compared with the help of extensionality.

Previous approaches to decide extensionality use eager rewriting [13], which blows up in space, as our experiments confirm. Our main contribution is a novel lemmas on demand algorithm for the extensional theory of arrays. We focus on

bit-vectors, but our approach can be easily generalized to other theories. We precisely describe our new propagation based algorithm for the non-extensional case, and generalize it to the extensional case. Finally, we report on experiments.

2 Background

The theory of bit-vectors allows precise modelling of actual computation in hardware and software. It models modular arithmetic, e.g. addition is modulo 2^{32}, using two's complement representation. Other operations include comparison and logical operations, shifting, concatenation, and bit extraction.

Arrays are one of the most important data structures in computer science [14]. The basic operations on arrays are *read*, *write* and equality on array *elements*. With $read(a, i)$ we denote the value of the array a at index i. With $write(a, i, e)$ we denote the array a, overwritten at index i with the new element e. All other array elements remain the same. From the theory of uninterpreted functions $\mathcal{EUF}$ we inherit the array congruence axiom:

$$\text{(A1)} \quad a = b \ \wedge \ i = j \quad \Rightarrow \quad read(a, i) = read(b, j)$$

Our approach does not need explicit congruence closure. The remaining non-extensional axioms of the theory of arrays are [15, 16]:

$$\text{(A2)} \quad i = j \quad \Rightarrow \quad read(write(a, i, e), j) = e$$
$$\text{(A3)} \quad i \neq j \quad \Rightarrow \quad read(write(a, i, e), j) = read(a, j)$$

Variables in these axioms are assumed to be universally quantified. Axiom (A2) asserts that the value read at an index is the same as the last value written to this index. Axiom (A3) asserts that writing to an index, does not change the values at other indices. In principle, axioms (A2) and (A3) can be used as rewrite rule, which allows to replace every read following a write by a *conditional* expression:

$$read(write(a, i, e), j) \qquad \text{is rewritten to} \qquad cond(i = j, e, read(a, j))$$

The expression $cond(c, s, t)$ returns s if the condition c is *true*, and t otherwise. The benefit of this *eager* approach is that writes are completely eliminated. The drawback is a quadratic blowup [1]. In addition to the array operations discussed so far, SMT solvers support *conditionals* on arrays, e.g. $cond(c, a, b)$, where a and b are arrays. In principle, using a fresh array variable d,

$$cond(c, a, b) \qquad \text{can be rewritten to} \qquad (c \rightarrow a = d) \wedge (\overline{c} \rightarrow b = d),$$

but this requires two more array equalities, which might be expensive to handle.

The *extensional* theory of arrays allows to compare arrays [15, 16]:

$$\text{(A4)} \quad a = b \quad \Leftarrow \quad \forall i \, (read(a, i) = read(b, i))$$

The other direction of the implication is already covered by (A1). The extensionality axiom (A4) is required for reasoning about array *inequalities*:

$$\text{(A4}') \quad a \neq b \quad \Rightarrow \quad \exists \lambda \, (read(a, \lambda) \neq read(b, \lambda))$$

3 Algorithm

We use lemmas on demand [8–12] for bit-vectors and one-dimensional[1] arrays, similar to [1], but for the extensional theory of arrays. We also describe the algorithm on a much more precise level.

Bit-vector variables and operations are eagerly encoded into SAT, including fresh variables for array equalities and read operations. Array equality and read are the only array operations which produce bit-vectors. The remaining array operations, write and conditional, are not encoded, as they return arrays. The resulting SAT instance can produce spurious satisfying assignments, invalid in the extensional theory of arrays. Therefore, if the SAT solver returns a satisfying assignment σ, then we still have to check all array axioms, before concluding satisfiability. If an array axiom is violated, we generate a symbolic bit-vector lemma that rules out this and similar assignments. In our implementation this lemma is incrementally added on the CNF level. No additional expressions are generated.

We show that the lemmas on demand approach [8] can be generalized to the extensional theory of arrays. Adding bit-vector lemmas on demand is sufficient.

```
procedure lemmas-on-demand (φ)
    encode-to-sat (φ)
    loop
        (r, σ) ← sat(φ)
        if (r = unsatisfiable) return unsatisfiable
        if (consistent (φ, σ)) return satisfiable
        add-lemma (φ, σ)
```

This algorithm [8] always terminates, as there are only finitely many assignments to bit-vector variables in ϕ. In every iteration the algorithm can terminate concluding unsatisfiability, or satisfiability if there is no theory inconsistency. However, if an inconsistency is detected, this and similar inconsistent assignments are ruled out, and the algorithm continues with the next refinement iteration.

4 Consistency Checker

Let ϕ denote the formula for which satisfiability is checked, and let σ denote a concrete assignment, generated by the SAT solver. We represent ϕ as an expression DAG. Expressions are either arrays or bit-vectors.

4.1 Read

If the only array operations are reads, the only array expressions in ϕ are array *variables*. The consistency checker iterates over all array variables and checks

[1] Multi-dimensional arrays can be represented by one-dimensional arrays.

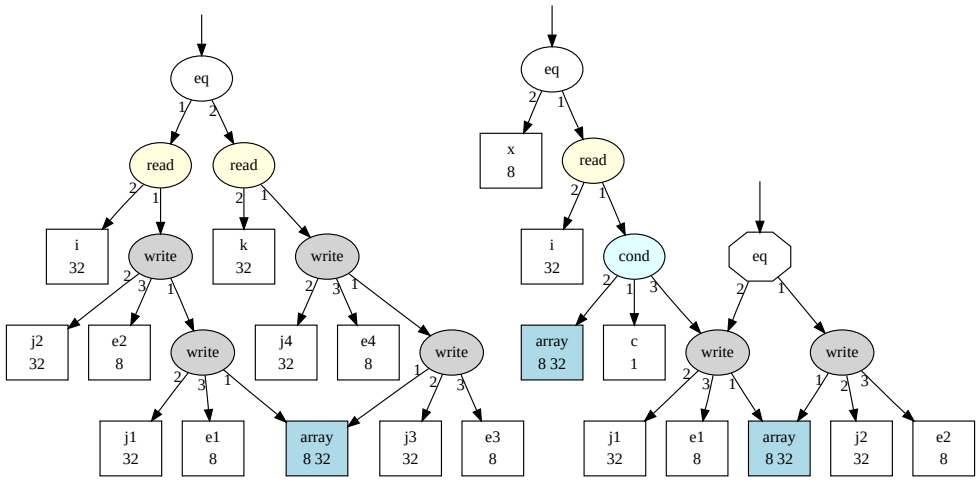

Fig. 1. ϕ for the examples 1 and 2 (left) and the multi-rooted examples 3 and 4 (right).

congruence. If (A1) is violated, i.e. $\sigma(i) = \sigma(j)$, but $\sigma(read(a, i)) \neq \sigma(read(a, j))$, a read-read conflict occurs, and the symbolic bit-vector lemma is added:

$$i = j \quad \Rightarrow \quad read(a, i) = read(a, j)$$

All lemmas are encoded directly on the CNF level. No additional expressions are generated. Due to space constraints, encoding details have to be skipped.

4.2 Write

If we also consider write expressions in ϕ, then an array expression is either an array variable or a write operation. Note that ϕ can contain *nested* writes. If array axiom (A2) or (A3) is violated, a read-write conflict occurs. In the first phase, the consistency checker iterates over array expressions and propagates reads:

1. Map all a to its set of reads $\rho(a)$, initialized as $\rho(a) = \{read(a, j) \text{ in } \phi\}$.
2. For all $read(b, i) \in \rho(write(a, j, e))$: if $\sigma(i) \neq \sigma(j)$, add $read(b, i)$ to $\rho(a)$.
3. Repeat step 2 until fix-point is reached, i.e. ρ does not change anymore.

In the second phase consistency is checked:

4. For all $read(b, i)$, $read(c, k)$ in $\rho(a)$: check *adapted* congruence axiom.
5. For all $read(b, i) \in \rho(write(a, j, e))$, $\sigma(i) = \sigma(j)$: check $\sigma(read(b, i)) = \sigma(e)$.

In step 4 the original array congruence axiom (A1) can not be applied, as $\rho(a)$ can contain propagated reads, which do not read on a directly. An adapted

axiom is violated if $\sigma(i) = \sigma(k)$, but $\sigma(\mathit{read}(b,i)) \neq \sigma(\mathit{read}(c,k))$. We collect all indices $j_1^i \dots j_m^i$, which have been used as j in the update rule for ρ (step 2) while propagating $\mathit{read}(b,i)$. Similarly, we collect all indices $j_1^k \dots j_n^k$ used as j in the update rule for ρ while propagating $\mathit{read}(c,k)$. We add the following symbolic bit-vector lemma:

$$i = k \ \wedge \bigwedge_{l=1}^{m} i \neq j_l^i \ \wedge \bigwedge_{l=1}^{n} k \neq j_l^k \quad \Rightarrow \quad \mathit{read}(b,i) = \mathit{read}(c,k)$$

The first big conjunction encodes that $\sigma(i)$ is different from all the assignments to the write indices on the propagation path of $\mathit{read}(b,i)$, the second conjunction that $\sigma(k)$ is different from all the write indices on the propagation path of $\mathit{read}(c,k)$. Adding this symbolic lemma makes sure that in the consistency checks of the following refinement iterations, the two reads can not produce the same conflict: either the propagation paths change, or the reads are congruent.

Example 1. Consider the reads $r_1 := \mathit{read}(\mathit{write}(\mathit{write}(a, j_1, e_1), j_2, e_2), i)$ and $r_2 := \mathit{read}(\mathit{write}(\mathit{write}(a, j_3, e_3), j_4, e_4), k)$, as shown in Fig. 1. Let ϕ be $r_1 = r_2$. We assume that the SAT solver has generated the following assignments: $\sigma(i) = 0$, $\sigma(k) = 0$, $\sigma(j_1) = 1$, $\sigma(j_2) = 5$, $\sigma(j_3) = 1$, $\sigma(j_4) = 2$, $\sigma(r_1) = 3$ and $\sigma(r_2) = 4$, i.e. all the write indices are different from the two identical read indices, and the read values differ. Read r_1 is propagated down to a, as $\sigma(i) \neq \sigma(j_2)$ and $\sigma(i) \neq \sigma(j_1)$. Read r_2 is propagated down in the same way. We check $\rho(a)$, find an inconsistency according to step 4, as $\sigma(i) = \sigma(k)$, but $\sigma(r_1) \neq \sigma(r_2)$, and add the following lemma:

$$i = k \ \wedge \ i \neq j_1 \ \wedge \ i \neq j_2 \ \wedge \ k \neq j_3 \ \wedge \ k \neq j_4 \quad \Rightarrow \quad r_1 = r_2$$

If the check in step 5 fails, i.e. axiom (A2) is violated, then we generate a similar lemma. In this case one of the propagation paths is empty.

Example 2. Let ϕ be as in example 1, and $\sigma(i) = 3$, $\sigma(r_1) = 1$, $\sigma(j_2) = 0$, $\sigma(j_1) = 3$ and $\sigma(e_1) = 4$. We propagate r_1 down to $\mathit{write}(a, j_1, e_1)$, as $\sigma(i) \neq \sigma(j_2)$. We check $\rho(\mathit{write}(a, j_1, e_1))$, find an inconsistency according to step 5, as $\sigma(i) = \sigma(j_1)$, but $\sigma(r_1) \neq \sigma(e_1)$, and add the following lemma:

$$i = j_1 \wedge i \neq j_2 \quad \Rightarrow \quad r_1 = e_1$$

Fix-point computation can be implemented as post-fix DFS traversal on the array expression sub graph of ϕ interleaved with on-the-fly consistency checking. Without array conditionals nor equality the sub graph is actually a tree, which simplifies this traversal further.

4.3 Conditional

As soon as we add conditionals on arrays, our array expressions become real DAGs. We add the following rule after step 2:

2*a*. For all $read(b,i) \in \rho(cond(c,t,e))$: if $\sigma(c) = 1$, add $read(b,i)$ to $\rho(t)$, else add $read(b,i)$ to $\rho(e)$.

The current assignment of the SAT solver determines which array is selected. If the condition is set to 1, reads are propagated down to t, otherwise down to e.

The lemma, generated upon inconsistency detection, is extended in the following way. We additionally collect all conditions $c_1^i, \ldots c_o^i$ used as c in step 2*a* while propagating down $read(b,i)$. Similarly, we collect all conditions $c_1^k, \ldots c_p^k$ used as c in step 2*a* while propagating down $read(c,k)$. Two more conjunctions are added:

$$\ldots \wedge \bigwedge_{l=1}^{o} c_l^i = \sigma(c_l^i) \wedge \bigwedge_{l=1}^{p} c_l^k = \sigma(c_l^k) \quad \Rightarrow \quad read(b,i) = read(c,k)$$

Note that on the CNF level, $c_k = \sigma(c_k)$ can be represented as one literal.

4.4 Equality

Finally, we also consider extensionality and introduce array equalities. For every array equality $a = b$ we generate a fresh boolean variable $e_{a,b}$, as in the Tseitin Transformation [17]. Two *virtual* reads, $read(a,\lambda)$ and $read(b,\lambda)$, over a fresh index variable λ, are added. Then we add as constraint:

$$\bar{e}_{a,b} \quad \Rightarrow \quad read(a,\lambda) \neq read(b,\lambda)$$

The idea of this constraint is that virtual reads are used as witness for array *inequality*. The SAT solver is only allowed to set $e_{a,b}$ to *false* if there is an assignment to λ, such that $read(a,\lambda) \neq read(b,\lambda)$. A similar usage of λ can be found in [16], and as k in rule *ext* [13]. To propagate reads over array equalities, we add rule 2*b*:

2*b*. For all $a = b$ where $\sigma(e_{a,b}) = 1$ and $read(c,i) \in \rho(a)$: add $read(c,i)$ to $\rho(b)$ and vice versa.

Rule 2*b* expresses that we also have to propagate reads over an array equality, but only if the SAT solver assigns it to *true*.

However, the changes made so far are not sufficient. Consider the formula:

$$write(a,i_1,e_1) = write(b,i_2,e_2) \wedge i_1 = i_2 \quad \Rightarrow \quad e_1 = e_2$$

So far this formula can not be shown to hold, although it is a theorem. To solve this problem, we interpret writes as reads and propagate them in the same way. As a result of this, we enforce that the write values e_1 and e_2 have to be equal, as writes interpreted as reads have to be congruent according to axiom (A1). Note that write propagation is only necessary as we do not handle $\mathcal{EUF}$ explicitly, i.e. congruence of *write*.

We use a polymorphic expression *access* in our implementation. An access expression can be a read or write. This simplifies the implementation, as we do not have to distinguish between read or write during propagation.

As soon as array equalities are introduced, fix-point computation is needed, as propagation paths are cyclic. A simple example is the transitivity of array equality. Recall that we introduce two virtual reads for every array equality, which in this example are propagated in a cyclic way. Hence, post-order traversal is no longer sufficient.

The lemma, generated upon inconsistency detection, is extended in the following way. We additionally collect all array equalities $e^i_1, \ldots e^i_q$ used as e in step 2b while propagating down $\mathit{read}(b, i)$. Similarly, we collect all array equalities $e^k_1, \ldots e^k_r$ used as e in step 2b while propagating down $\mathit{read}(c, k)$. Two more conjunctions are added:

$$\ldots \wedge \bigwedge_{l=1}^{q} e^i_l \wedge \bigwedge_{l=1}^{r} e^k_l \quad \Rightarrow \quad \mathit{read}(b, i) = \mathit{read}(c, k)$$

Example 3. Now consider $w_1 := \mathit{write}(a, j_1, e_1)$, $r := \mathit{read}(\mathit{cond}(c, b, w_1), i)$ and $w_2 := \mathit{write}(a, j_2, e_2)$, as shown in Fig. 1. Let ϕ be $r = x \wedge w_1 = w_2$. We assume that the SAT solver has generated the following assignments: $\sigma(j_1) = 0$, $\sigma(j_2) = 0$, $\sigma(e_1) = 0$, $\sigma(e_2) = 3$ and $\sigma(e_{w_1,w_2}) = 1$. We propagate w_2 as read over the array equality $w_1 = w_2$ to w_1. We find an inconsistency for $\rho(w_1)$, according to step 4, as $\sigma(j_1) = \sigma(j_2)$, but $\sigma(e_1) \neq \sigma(e_2)$, and add the lemma:

$$j_1 = j_2 \wedge e_{w_1,w_2} \Rightarrow e_1 = e_2$$

Example 4. Now let ϕ be as in example 3, and $\sigma(i) = 0$, $\sigma(r) = 1$, $\sigma(c) = 0$, $\sigma(j_1) = 1$, $\sigma(j_2) = 0$, $\sigma(e_2) = 5$ and $\sigma(e_{w_1,w_2}) = 1$. We propagate r down to a, as $\sigma(c) = 0$ and $\sigma(i) \neq \sigma(j_1)$. We propagate w_2 as read over e_{w_1,w_2} to w_1 and down to a, as $\sigma(e_{w_1,w_2}) = 1$ and $\sigma(j_1) \neq \sigma(j_2)$. We find an inconsistency for $\rho(a)$, according to step 4, as $\sigma(i) = \sigma(j_2)$, but $\sigma(r) \neq \sigma(e_2)$, and add the following lemma:

$$i = j_2 \ \wedge \ i \neq j_1 \ \wedge \ j_2 \neq j_1 \ \wedge \ c = 0 \ \wedge \ e_{w_1,w_2} \quad \Rightarrow \quad r = e_2$$

The rules presented so far, do not propagate upwards, e.g. axiom (A3) is only applied from left to right. Consequently, there is one class of formulas, where the consistency checker fails to find inconsistencies. Consider the following example:

$$\mathit{write}(a, i, e_1) = \mathit{write}(b, j, e_2) \ \wedge \ i \neq k \ \wedge \ j \neq k \ \wedge \ \mathit{read}(a, k) \neq \mathit{read}(b, k)$$

This formula is unsatisfiable, which can be derived by applying axiom (A1) and (A3). The access objects are not propagated upwards, therefore the reads are never propagated to the same array expression. The inconsistency can not be detected by the set of rules discussed so far. To solve this problem, we finally complete our consistency checking algorithm as follows:

2*c*. For all $read(b,i) \in \rho(a)$: if $\sigma(i) \neq \sigma(j)$, add $read(b,i)$ to $\rho(write(a,j,e))$.
2*d*. For all $read(b,i) \in \rho(t)$: if $\sigma(c) = 1$, add $read(b,i)$ to $\rho(cond(c,t,e))$.
2*e*. For all $read(b,i) \in \rho(e)$: if $\sigma(c) = 0$, add $read(b,i)$ to $\rho(cond(c,t,e))$.

4.5 Complexity, Soundness and Completeness

The consistency checking algorithm has a worst case quadratic complexity. More specifically the number of expression nodes $|\phi|$ is an upper bound on the number of read expression nodes and also an upper bound on the number of array expression nodes. Therefore there are at most $O(|\phi|^2)$ updates to ρ in one run of the consistency checker.

Soundness follows from the fact, that all our rules respect axioms. Thus, if the consistency checker finds a conflict, then the current assignment does not satisfy the extensional theory of arrays. Our rules simply model all ways of applying axioms (A1) to (A4). Thus, if our consistency checker determines consistency, the model is also consistent in the extensional theory of arrays.

5 Experimental Evaluation

We implemented our lemmas on demand algorithm for the extensional theory of arrays in our new SMT solver Boolector[2], including the previous state of the art decision procedure [13]. Our implementation of [13] eliminitates equalities on arrays by eager rewriting. Side conditions of rewrite rules are handled symbolically to avoid case splitting.

Boolector uses a functional AIG encoding with two level AIG rewriting [18], as in [19]. Picosat [20] is used as SAT solver.

We ran our benchmarks on our cluster of 3 GHz Pentium IV with 2 GB main memory, running Ubuntu Linux. We set a time limit of 900 seconds and a memory limit of 1500 MB.

Our set of benchmarks, see Table 1, contains extensional examples, where computer memory is modelled as one big array of 2^{32} bytes. Benchmark `swapmem` swaps with a combination of XOR operations two byte sequences in memory twice. Extensionality is used to show that the final memory is equal to the initial. An instance is satisfiable if the sequences can overlap, and unsatisfiable otherwise. Benchmark `dubreva` reverses multiple byte sequences in memory twice. Again, extensionality is used to show that the final memory is equal to the initial. The instances are all unsatisfiable. In benchmarks `wchains` we check the following. Given P 32 bit pointers, the 32 bit word a pointer points to is overwritten with its pointer address. Then, the order in which the pointers are written does not matter as long the pointers are 4 byte aligned.

[2] http://fmv.jku.at/boolector

			lemmas			*eager*		z3		cvc3	
benchmark	S	P	R	T	M	T	M	T	M	T	M
swapmem	s	2	16	0.1	0.6	**0.0**	**0.0**	0.1	5.0	554.1	18.2
swapmem	s	6	53	**0.4**	**1.2**	0.6	3.3	547.6	101.6	90.0	71.9
swapmem	s	8	37	**0.3**	**1.4**	1.5	5.4	out of time	out of time	333.8	175.8
swapmem	s	10	34	**0.5**	**1.7**	1.0	8.3	out of time	out of time	740.2	305.3
swapmem	s	12	64	**1.0**	**2.2**	3.5	11.2	out of time	out of time	out of time	out of time
swapmem	s	14	57	**1.6**	**2.4**	1.8	15.7	out of time	out of time	out of time	out of time
wchains	s	6	14	**0.0**	**0.0**	0.8	7.0	46.3	11.6	8.5	60.1
wchains	s	8	16	**0.0**	**0.0**	1.9	14.5	572.2	32.5	23.5	117.6
wchains	s	10	18	**0.1**	**1.3**	3.1	20.3	out of time	out of time	34.2	193.6
wchains	s	12	20	**0.1**	**1.4**	3.5	29.8	out of time	out of time	65.4	294.6
wchains	s	15	21	**0.1**	**1.5**	4.2	45.8	out of time	out of time	126.3	472.6
wchains	s	20	28	**0.3**	**2.0**	5.7	80.2	out of time	out of time	267.8	897.3
wchains	s	30	36	**0.6**	**2.8**	10.9	179.9	out of time	out of time	out of memory	out of memory
wchains	s	60	70	**2.3**	**4.8**	41.8	718.5	out of time	out of time	out of memory	out of memory
wchains	s	90	96	**4.8**	**6.8**	out of memory	out of memory	out of time	out of time	out of memory	out of memory
dubreva	u	2	19	0.1	0.6	**0.0**	**0.0**	0.5	5.0	out of time	out of time
dubreva	u	3	41	0.3	**0.7**	**0.1**	1.2	0.7	5.6	out of time	out of time
dubreva	u	4	239	**12.5**	**2.2**	**12.5**	6.5	out of time	out of time	out of time	out of time
dubreva	u	5	379	**27.4**	**3.7**	348.3	13.1	out of time	out of time	out of time	out of time
dubreva	u	6	1187	**322.2**	**12.4**	out of time	out of time	out of time	out of time	out of time	out of time
dubreva	u	7	1637	**663.1**	**18.2**	out of time	out of time	out of time	out of time	out of memory	out of memory
swapmem	u	2	13	**0.1**	**0.7**	**0.1**	0.8	1.2	5.5	6.1	9.0
swapmem	u	4	25	**1.1**	**1.0**	3.0	2.0	5.2	7.1	159.6	49.8
swapmem	u	6	37	**3.2**	**1.2**	27.9	4.2	16.1	8.6	out of time	out of time
swapmem	u	8	49	**9.0**	**1.5**	129.0	8.3	20.7	10.1	out of time	out of time
swapmem	u	10	61	**18.3**	**2.0**	411.2	15.4	30.9	11.0	out of time	out of time
swapmem	u	12	73	**34.7**	**2.4**	out of time	out of time	62.1	13.3	out of time	out of time
swapmem	u	14	85	**63.4**	**2.8**	out of time	out of time	102.7	19.4	out of time	out of time
wchains	u	2	17	**0.0**	**0.0**	**0.0**	**0.0**	out of time	out of time	4.1	5.8
wchains	u	4	33	**0.1**	**0.9**	1.7	3.8	10.5	16.4	out of time	out of time
wchains	u	6	49	**0.4**	**1.1**	13.5	8.2	1.9	5.5	out of memory	out of memory
wchains	u	8	65	**0.6**	**1.3**	54.8	15.0	848.0	725.1	out of memory	out of memory
wchains	u	10	81	**1.1**	**1.5**	158.2	23.3	59.7	14.6	out of memory	out of memory
wchains	u	12	97	**2.4**	**1.8**	333.5	34.3	93.7	22.4	out of memory	out of memory
wchains	u	14	113	**2.9**	**1.9**	588.2	46.7	19.3	6.9	out of memory	out of memory
wchains	u	16	129	**4.8**	**2.1**	out of time	out of time	35.7	8.1	out of memory	out of memory
wchains	u	18	145	**6.6**	**2.3**	out of time	out of time	209.8	26.2	out of memory	out of memory

Table 1. *Extensional Examples*: We compare our lemmas on demand approach with (i) the eager approach [13], implemented as rewrite system, (ii) Z3 1.2 [5], and (iii) CVC3 1.2.1 [4]. The first column gives the benchmark name, the second the satisfiability S, followed by the benchmark parameter P. In the fourth column, labelled R, the number of refinements (lemmas on demand) is shown. Columns T and M show solving time in seconds and memory usage in MB. Our approach clearly outperforms all the others.

6 Conclusion

We showed that lemmas on demand can also handle the extensional theory of arrays. Our key contributions are a *precise* formulation of a propagation based approach and its *generalization* to extensionality. Our implementation shows very competitive performance. Future optimizations include a linear equation solver [1] and a faster rewriting engine. Finally, we want to thank Nikolaj Bjørner and Leonardo De Moura for their helpful comments on an earlier version of this paper.

References

1. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: Proc. CAV. (2007)
2. Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., Hanna, Z., Nadel, A., Palti, A., Sebastiani, R.: A lazy and layered SMT($\mathcal{BV}$) solver for hard industrial verification problems. In: Proc. CAV. (2007)
3. Manolios, P., Srinivasan, S., Vroon, D.: BAT: The bit-level analysis tool. In: Proc. CAV. (2007)
4. Barrett, C., Tinelli, C.: CVC3. In: Proc. CAV. (2007)
5. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Proc. TACACS. (2008)
6. Nieuwenhuis, R., Oliveras, A.: Decision procedures for SAT, SAT modulo theories and beyond. the BarcelogicTools. In: Proc. LPAR. (2005)
7. Sebastiani, R.: Lazy satisfiability modulo theories. JSAT **3** (2007)
8. de Moura, L., Rueß, H.: Lemmas on demand for satisfiability solvers. In: Proc. SAT. (2002)
9. Flanagan, C., Joshi, R., Ou, X., Saxe, J.B.: Theorem proving using lazy proof explication. In: Proc. CAV. (2003)
10. Barrett, C., Dill, D.L., Stump, A.: Checking satisfiability of first-order formulas by incremental translation to SAT. In: Proc. CAV. (2002)
11. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. In: Proc. BMC. (2003)
12. Bryant, R., Kroening, D., Ouaknine, J., Seshia, S., Strichman, O., Brady, B.: Deciding bit-vector arithmetic with abstraction. In: Proc. TACAS. (2007)
13. Stump, A., Barrett, C., Dill, D.L., Levitt, J.R.: A decision procedure for an extensional theory of arrays. In: Proc. LICS. (2001)
14. McCarthy, J.: Towards a mathematical science of computation. In: Proc. IFIP Congress. (1962)
15. Bradley, A., Manna, Z., Sipma, H.: What's decidable about arrays? In: Proc. VMCAI. (2006)
16. Bradley, A., Manna, Z.: The Calculus of Computation - Decision Procedures with Applications to Verification. Springer (2007)
17. Tseitin, G.: On the Complexity of Derivation in Propositional Calculus. In: Studies in Constructive Mathematics and Mathematical Logic, Part II. (1968)
18. Brummayer, R., Biere, A.: Local two-level and-inverter graph minimization without blowup. In: Proc. MEMICS. (2006)
19. Brummayer, R., Biere, A.: C32SAT: Checking C expressions. In: Proc. CAV. (2007)
20. Biere, A.: PicoSAT essentials. JSAT (2008) submitted.

Towards SMT Model Checking of Array-based Systems*

Silvio Ghilardi[1], Enrica Nicolini[2], Silvio Ranise[3], and Daniele Zucchelli[1]

[1] Dipartimento di Informatica, Università degli Studi di Milano (Italia)
[2] LORIA & INRIA-Lorraine, Nancy (France)
[3] Dipartimento di Informatica, Università di Verona (Italia)

Abstract. We introduce the notion of array-based system as a suitable abstraction of infinite state systems such as broadcast protocols or sorting programs. By using a class of quantified-first order formulae to symbolically represent array-based systems, we propose methods to check safety (invariance) and liveness (recurrence) properties on top of Satisfiability Modulo Theories solvers. We find hypotheses under which the verification procedures for such properties can be fully mechanized.

* This paper appears in the Proc. of IJCAR'08.

Back to the Future
Revisiting Precise Program Verification using SMT Solvers

Shuvendu K. Lahiri, Shaz Qadeer

Microsoft Research, Redmond, USA
{shuvendu, qadeer}@microsoft.com

Abstract. This paper takes a fresh look at the problem of precise verification of heap-manipulating programs using first-order Satisfiability-Modulo-Theories (SMT) solvers. We augment the specification logic of such solvers by introducing the Logic of Interpreted Sets and Bounded Quantification for specifying properties of heapmanipulating programs. Our logic is expressive, closed under weakest preconditions, and efficiently implementable on top of existing SMT solvers. We have created a prototype implementation of our logic over the solvers SIMPLIFY and Z3 and used our prototype to verify many programs. Our preliminary experience is encouraging; the completeness and the efficiency of the decision procedure is clearly evident in practice and has greatly improved the user experience of the verifier.[1]

[1] This paper appears in the Proc. of POPL'08.

SMELS: Satisfiability Modulo Equality with Lazy Superposition

Christopher Lynch[1] and Duc-Khanh Tran[2]

[1] Clarkson University, USA
[2] Max-Planck-Insitut für Informatik, Germany

Abstract. We give a method for extending efficient SMT solvers to handle quantifiers, using Superposition inference rules. In our method, the input formula is converted into CNF as in traditional first order logic theorem provers. The ground clauses are given to the SMT solver, which runs a DPLL method to build partial models. The partial model is passed to a Congruence Closure procedure, as is normally done in SMT. Congruence Closure calculates all reduced (dis)equations in the partial model and passes them to a Superposition procedure, along with a justification. The Superposition procedure then performs an inference rule, which we call Justified Superposition, between the (dis)equations and the nonground clauses, plus usual Superposition rules with the nonground clauses. Any resulting ground clauses are provided to the DPLL engine. We prove the completeness of this method, using a nontrivial modification of Bachmair and Ganzinger's model generation technique. We believe this combination uses the best of both worlds, an SMT process to handle ground clauses efficiently, and a Superposition procedure which uses orderings to handle the nonground clauses.

Deciding Array Formulas with Frugal Axiom Instantiation

Amit Goel[1], Sava Krstić[1], and Alexander Fuchs[2]

[1] Strategic CAD Labs, Intel Corporation
[2] The University of Iowa

Abstract. How to efficiently reason about arrays in an automated solver based on decision procedures? The most efficient SMT solvers of the day implement "lazy axiom instantiation": treat the array operations read and write as uninterpreted, but supply at appropriate times appropriately many—not too many, not too few—instances of array axioms as additional clauses. We give a precise account of this approach, specifying "how many" is enough for correctness, and showing how to be frugal and correct.

1 Introduction

Suppose a theory $\mathcal{T}$ talks about some operations $f_1, \ldots, f_n$, specified axiomatically. If a query (set of quantifier-free formulas) Φ of this theory is unsatisfiable, then there exists a finite set L of axiom instances such that $\Phi \cup L$ is unsatisfiable in the "empty theory" where nothing is assumed about the f_i except that they are functions of the appropriate type. This existence is generally non-constructive (classical undecidability), but for some theories we are in luck. Kapur and Zarba [12] show that, barring integers, theories of the commonly used data types allow equisatisfiable "reduction" of this kind. In particular, a simple reduction to the theory of uninterpreted functions works for the McCarthy theory of arrays:

$$\mathsf{read}(\mathsf{write}(a,i,x),j) \;=\; \begin{cases} x & \text{if } i = j \\ \mathsf{read}(a,j) & \text{otherwise} \end{cases} \qquad \text{(read-over-write)}$$

$$a \neq b \;\Rightarrow\; \exists i.\, \mathsf{read}(a,i) \neq \mathsf{read}(b,i) \qquad \text{(extensionality)}$$

The array reduction procedure in [12] is pleasingly simple to describe and verify, but not suitable for direct implementation because of a large number of unnecessary axiom instances it introduces. Of the few existing solvers that handle array benchmarks, the most efficient ones are indeed careful to generate axiom instances in a lazy fashion [10]. However, their exact algorithms are unspecified, let alone proved correct. We fill this gap by contributing a decision procedure with frugal axiom instantiation and proving it correct.

A proper setup to state and prove results of this kind requires a precise enough concept of a theory solver. This is a non-trivial issue in itself, since modern SMT solvers—to which we would like our results to be clearly applicable—combine a SAT solver and several theory solvers in architectures with complex

flow of data. In §2, we define a view of solvers that captures axiom instantiation, abstracting away other features that, even if indispensable for combination, can be made opaque without loss of precision and generality. In §3, we define a *theory of updatable functions*, a simple extension of the "theory of uninterpreted functions" into which array formulas readily translate. Then we give rule-based descriptions of three progressively finer solvers for this theory and briefly discuss their frugality. Implementation is discussed in §4. Correctness proofs and additional comments on the rules can be found online, in the extended version of the paper: `www.cs.uiowa.edu/~fuchs/PAPERS/array_solver.pdf`.

Related Work. The interaction between a SAT solver and a theory solver that creates axiom instances (and other lemmas) was first formalized by Barrett et al. [4]; our alternative in §2.2 is closer to the implementation level and directly applicable. Solving formulas about arrays started with the venerable *Simplify* [9] and its sophisticated quantifier instantiation mechanism; it is generic and thus suitable for any axiomatic theory, but it does not guarantee completeness. UCLID is complete for the non-extensional array fragment [8]. The first complete procedures were implemented in solvers of the CVC family [3], based on the theoretical work by Stump et al. [19]. Their rule-based system is more elaborate and a quick comparison of frugality is hardly possible. Armando et al. [2] construct a complete procedure from axioms by using superposition-based rewriting techniques, but do not look for optimizations. Dutertre and de Moura's solver *Yices* [10] deals with arrays by controlled axiom instantiation, the details of which are unavailable. Recent papers by Bradley et al. [7] and Ghilardi et al. [11] on array procedures focus on extending the language beyond the standard read/write; they do use axiom instantiation, just without emphasis on frugality.

2 Solvers

We adopt the typed setting as in [13], where we refer for precise definitions of *signatures*; of *types*, *terms*, *formulas* associated with each signature; and of *theories* over a given signature. Let us recall here that parametric types (involving type variables) are allowed; that the concrete type Bool and logical symbols including equality are implicitly present in every signature, and that formulas are just terms of type Bool. Given a theory $\mathcal{T}$ and a set of formulas Φ over its signature, we say that Φ is *satisfiable* if it has a *model*. The *$\mathcal{T}$-validity* $\models_{\mathcal{T}} \Phi$ means that the negation of Φ is unsatisfiable. Again, refer to §2 of [13] for full definitions.

Define $\phi \lll \psi$ to mean $\models \phi \Leftrightarrow (\exists \vec{y})\psi$, where $\vec{y}$ is the string of variables that occur in ψ, but not in ϕ. This *equisatisfiable expansion* relation is a strong form of equisatisfiability: every model of ψ restricts to a model of ϕ, and every model of ϕ extends to a model of ψ. Generalize to $\phi \lll_{\mathcal{T}} \psi$, standing for $\models_{\mathcal{T}} \phi \Leftrightarrow (\exists \vec{y})\psi$.

Define a *defset* (shorthand for "definitional set of equations") to be a set of the form $\{u_1 = e_1, \ldots, u_n = e_n\}$, where the u_i are distinct variables and the e_i are terms such that u_i occurs in e_j only if $j > i$. We call u_i the *proxy* for e_i.

For every quantifier-free formula ϕ over the union signature $\Sigma_1 + \cdots + \Sigma_n$, we have $\phi \lll \phi_0 \wedge \phi_1 \wedge \cdots \wedge \phi_n$, where ϕ_0 is a set of propositional clauses and each ϕ_i ($i \geq 1$) is a defset over Σ_i. Purification algorithms as in [18, 13] compute $\phi_0, \ldots, \phi_n$ from a given ϕ.

2.1 Combined SMT Solvers

Figure 1 gives a high-level picture of a modern SMT solver, showing the key state components of the abstract system NODPLL [13], which itself is a generalized, simplified, and elaborated version of the abstract DPLL($\mathcal{T}$) framework [17].

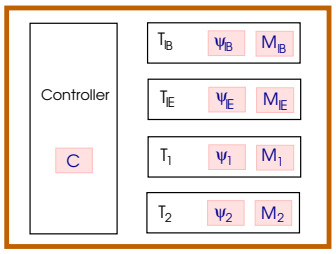

Fig. 1: The state and implementation modules of an SMT solver.

It comprises the solvers $T_{\mathbb{B}}$ and $T_{\mathbb{E}}$ for booleans and for equality respectively, and solvers $T_1, T_2, \ldots$ for other theories. The communication between solvers goes mainly by exchange of *interface literals*: propositional literals and (dis)equalities between variables. In a simple scenario, the set of literals is fixed at the initialization time, when the solvers are given their defsets Ψ_i obtained by purifying the mixed input formula, and their *literal stacks* M_i are all empty. If T_i can infer a new literal as a consequence of Ψ_i and M_i, it can put it on its stack M_i, and then the same literal can be *propagated* onto the other stacks M_j as well. When no T_i can make progress by such inference, the SAT solver $T_{\mathbb{B}}$ will put a new literal on its stack speculatively (*decision*), initiating thus a new round of inferences by itself and other solvers (new *decision level*). When one of the solvers detects that its logical state $M_i \wedge \Psi_i$ is inconsistent, it sets the *conflict set* C to a subset of M_i that contradicts Ψ_i. At this point we know that the sequence of speculative decisions is inconsistent and we need to backtrack. A sequence of *explanation* calls to the solvers ensues, where at each step C is modified into another set of literals that is inconsistent with the union of the Ψ_i. The set C remains a subset of the union of the M_i but gets progressively pushed back in time. This *conflict analysis* process results in finding an earlier decision level that is logically inconsistent with our last speculative decision. Then every solver *backjumps* to that decision level, a new literal (the negation of some literal from the current level) is deduced, and we continue as before. We are done either when a conflict is detected at level 0 (inconsistency), or there are no more literals to assert speculatively (model exists).

Completeness of the combined solver is a complex matter. By a classical result of Nelson and Oppen, completeness is guaranteed if all participating theories are convex and their solvers propagate every implied equality between shared non-boolean variables [16]. It is guaranteed also if we introduce proxy boolean variables for every equality between shared non-boolean variables. This is the method of *delayed theory combination* of Bozzano et al. [6, 18]. It relieves the

solvers from the obligation to propagate equalities and the theories do not need to be convex, but it comes with the cost of extensive proxying which forces the SAT solver to split over too many variables. The *splitting-on-demand* framework of Barrett et al. [4] allows theory solvers to introduce proxied equalities as needed and, more generally, to introduce new variables and communicate facts about them ("lemmas") to the system. Solving by axiom instantiation fits naturally into this framework, with axiom instances being the lemmas.

2.2 Theory Solvers

Starting to pin down the requirements on theory solvers that include mechanisms for adding new variables and lemmas, let us view the *state* of a solver abstractly as a sextuple:

V	a set of variables; V_{Bool} denoting its subset of boolean variables
Ψ	a defset with all variables in V
M	a partial assignment sequence over V_{Bool}
L	a set of clauses over V_{Bool} such that $\Psi \models_{\mathcal{T}} \phi$ for every $\phi \in L$
status	NO_CFLCT or CFLCT
local	solver-specific state

We will use the record notation $s.V$, $s.\Psi$,..., $s.\mathsf{local}$ to refer to components of a particular state s. In addition, we assume there is a function Λ that defines the *logical state* $\Lambda(s)$—the formula represented by the state s. It is the conjunction of $s.\Psi$, $s.M$, and a formula derived in the solver-specific manner from $s.\mathsf{local}$.

Each solver is specified for a fragment $\mathcal{F}$ of a specific theory $\mathcal{T}$. The theory-specific heart of the solver is abstracted in the local component of the state and in the state changes that modify it. We make only these general requirements:

(i) if a state-to-state transition modifies M, then all it does is addition of a literal, exactly as described by the rule Literal in Figure 2 below;

(ii) given any defset Ψ_0 in the fragment $\mathcal{F}$, there exists a well-defined *initial state* $s_{\text{init}}(\Psi_0)$ such that $\Psi_0 \lll_{\mathcal{T}} \Lambda(s_{\text{init}}(\Psi_0))$;

(iii) $\Lambda(s) \lll_{\mathcal{T}} \Lambda(s')$, for transitions $s \to s'$ of the solver that do not modify M.

Regarding **(i)**, notice that actual solvers (as conveyed in §2.1) get to know new literals either by being told, or by their own inference; the abstract view we adopt here conveniently obscures the difference between the two. Notice also that our abstract model goes astray from §2.1 by not allowing (dis)equalities as interface literals (members of M). This is done for convenience and the loss of generality is small; all proofs would extend without difficulty to the more general setting. Finally, the solver's activities related to conflict analysis and backtracking are all ignored as not pertaining to this formalization.

A state s will be called *conflicting* or *non-conflicting* depending on whether $s.\mathsf{status}$ is CFLCT or NO_CFLCT. A state is *final* if it is reachable from an initial state and there are no transitions from it. A solver is *sound* if the existence of a run that begins with $s_{\text{init}}(\Psi_0)$ and ends in a conflicting final state

s implies that $\Psi_0 \wedge s.M$ is $\mathcal{T}$-unsatisfiable. A solver is *complete* if the existence of a run that begins with $s_{\text{init}}(\Psi_0)$ and ends in a non-conflicting final state s with $s.L \wedge s.M$ (propositionally) satisfiable, implies that $\Psi_0 \wedge s.M$ is $\mathcal{T}$-satisfiable. The following lemma expressing soundness and completeness in terms of final states is a consequence of assumptions **(i-iii)**.

Lemma 1. *(a) A solver is complete if $\Lambda(s)$ is $\mathcal{T}$-satisfiable for every non-conflicting final state s for which $s.L \wedge s.M$ is (propositionally) satisfiable.*
(b) A solver is sound when $\Lambda(s)$ is $\mathcal{T}$-unsatisfiable for every conflicting state s.

The underlined phrases indicate an important subtlety. If we remove them from the text, we will still have a formally correct definition of a solver's completeness, and (in Lemma 1(a)) a sufficient condition for it. But that condition could be too strong! Our formulation expresses a natural expectation by the solver of its environment: heed the lemmas you are getting from us so when you give us a new literal we can trust that none of our lemmas is violated. The completeness proof of the array solver described in §3 below will go by checking the condition in Lemma 1(a) and would not work with the stronger condition.

Having weakened the concept of individual solvers' completeness, we need to justify that it is still strong enough to guarantee completeness of the combined solver. This requires the additional assumption that the SAT solver does add all the generated lemmas to its clause base. A complete argument would need a precise definition of the combined solver and can be formulated without difficulty within the NODPLL system [13].

3 A Solver for Arrays as Updatable Functions

The *parametric theory of uninterpreted functions* [14] has a signature consisting of the type constructor $\Rightarrow$, interpreted as the function space operator, and the function symbol $@^{[\alpha\Rightarrow\beta,\alpha]\to\beta}$ interpreted as function application. The *theory $\mathcal{U}$ of updatable functions* is obtained by adding to this the symbol $U^{[\alpha\Rightarrow\beta,\alpha,\beta]\to(\alpha\Rightarrow\beta)}$ whose meaning is the update operator: $U(a,i,x)$ is the function b such that $b@i = x$ and $b@j = a@j$ for $j \neq i$.[3] Since arrays can be viewed as functions, with operations read and write seen as $@$ and U, solvers for $\mathcal{U}$ are solvers for the array read/write theory as well. The application symbol will be omitted; we will simply write ai instead of $a@i$.

For every $\mathcal{U}$-defset Ψ there is a defset Ψ' such that $\Psi \lll \Psi'$ and Ψ' is "flattened" so that all its equations have one of the following forms:

$$p \triangleq (u = v) \qquad x \triangleq ai \qquad b \triangleq U(a,i,x) \tag{1}$$

Read $\triangleq$ here just as $=$; the little triangle is just to remind us that the equality is definitional. We will assume that Bool is the only concrete type used, and that its use is limited to range positions (no type of a variable is allowed to contain

[3] The theory $\mathcal{U}$ can be easily extended with the *constant function symbol* $K^{\beta\to(\alpha\Rightarrow\beta)}$ interpreted as $K(x) = \lambda i.x$. Only for simplicity, we restrict ourselves to $\mathcal{U}$.

$\mathsf{Bool} \Rightarrow \sigma$ as a subexpression). It is well-known that without the last assumption, or some similar condition, even the congruence closure algorithm would not be complete. (See the discussion about *cardinality constraints* in [14].)

We proceed to describe three related solvers for $\mathcal{U}$, named UPD_0, UPD_1 and UPD_2, that conform to the requirements set in §2.2. The solvers' defsets are of the form (1), and their only local state component is an equivalence relation $\sim$ on the set $V - V_{\mathsf{Bool}}$ of non-boolean variables. The $\sim$-equivalence class of u will be denoted $[u]$. We will write $\sim_s$ when referring to $\sim$ at a given state s. By definition, the theory-specific contribution to the logical state function $\Lambda(s)$ is the conjunction of all equalities of the form $u = v$, where $u \sim_s v$. Recall that, in addition to this, $\Lambda(s)$ contains $s.M$ and $s.\Psi$ as conjuncts. Intuitively, $u \sim_s v$ means "u and v are known to be equal in the state s", i.e., it is an invariant that $u \sim_s v$ implies $\Lambda(s) \models_{\mathcal{U}} u = v$.

Define $a \rtimes_i b$ to mean that for some x, the equation $a \triangleq U(b, i, x)$ is in Ψ. The equivalence relation generated by $\sim$ together with all relations $\rtimes_i$ will be denoted $\bowtie$. The relations $\rtimes_i$ will not change from state to state, but $\bowtie$ will. Intuitively, if $a, b \in V^{\sigma \Rightarrow \tau}$, then $a \bowtie_s b$ implies that in the state s we know that a and b agree on all but finitely many arguments.

Define $\mathsf{proxied}_s(e)$ to mean that e occurs in some proxy equation $(u \triangleq e) \in s.\Psi$, in which case we also write $\lceil e \rceil_s = u$. Generalizing this to arbitrary terms, define $\lceil e \rceil_s$ to be the term obtained from e by recursively replacing subterms with their $s.\Psi$-proxies until there are no $\mathsf{proxied}_s$-subterms anymore. The set $s.\Psi$ of proxy equations will monotonically increase from state to state. We will suppress the subscript s from $\mathsf{proxied}_s$ and $\lceil e \rceil_s$ when it is clear from the context.

The transitions for our solvers are given by the rules in Figure 2. Only the rules Eq and Congr modify the local state $\sim$; together with Conflict, they give an abstract presentation of the congruence-closure algorithm. The state-transforming function $\mathsf{proxy}(e)$ used in the remaining rules introduces a proxy variable for every application and equality subterm of e, if it does not already exist, and returns the final boolean combination of propositional variables. For example, the action $L := L + \mathsf{proxy}(i \neq j \Rightarrow aj = bj)$ in $\mathsf{RoW}_{0,1,2}$ means addition of some of the equations $x \triangleq aj$, $y \triangleq bj$, $p \triangleq (i = j)$, and $q \triangleq (x = y)$ to Ψ with fresh variables x, y, p, q, followed by addition of the clause $p \vee q$ to L. The criterion for adding $x \triangleq ai$ to Ψ (and x to V) is that Ψ does not already contain a proxy for ai; if it does, then x just denotes that proxy. In Ext_arg, the action $\mathsf{proxy}(a = b)$ means adding $p \triangleq (a = b)$ to Ψ if the proxy for $a = b$ does not exist in Ψ. The occurrences of proxy in $\mathsf{Ext}_{0,1,2}$ are to be understood analogously.

We use the convention that the same rule with the same parameters cannot fire twice. For Literal and Conflict this is already true, but guards of the other rules need an additional progress condition. For Eq and Congr these conditions are $u \not\sim v$ and $\lceil ai \rceil \not\sim \lceil bj \rceil$ respectively. For $\mathsf{RoW}_{0,1,2}$, the progress means that L does not contain $\lceil i \neq j' \Rightarrow aj' = bj' \rceil$ for any $j' \sim j$, and for $\mathsf{Ext}_{0,1,2}$ the condition is that L does not contain $\lceil a' \neq b' \Rightarrow a'i \neq b'i \rceil$ for any i and any $a' \sim a$ and $b' \sim b$. Finally, the progress condition for Ext_arg is $\neg\mathsf{proxied}(a' = b')$

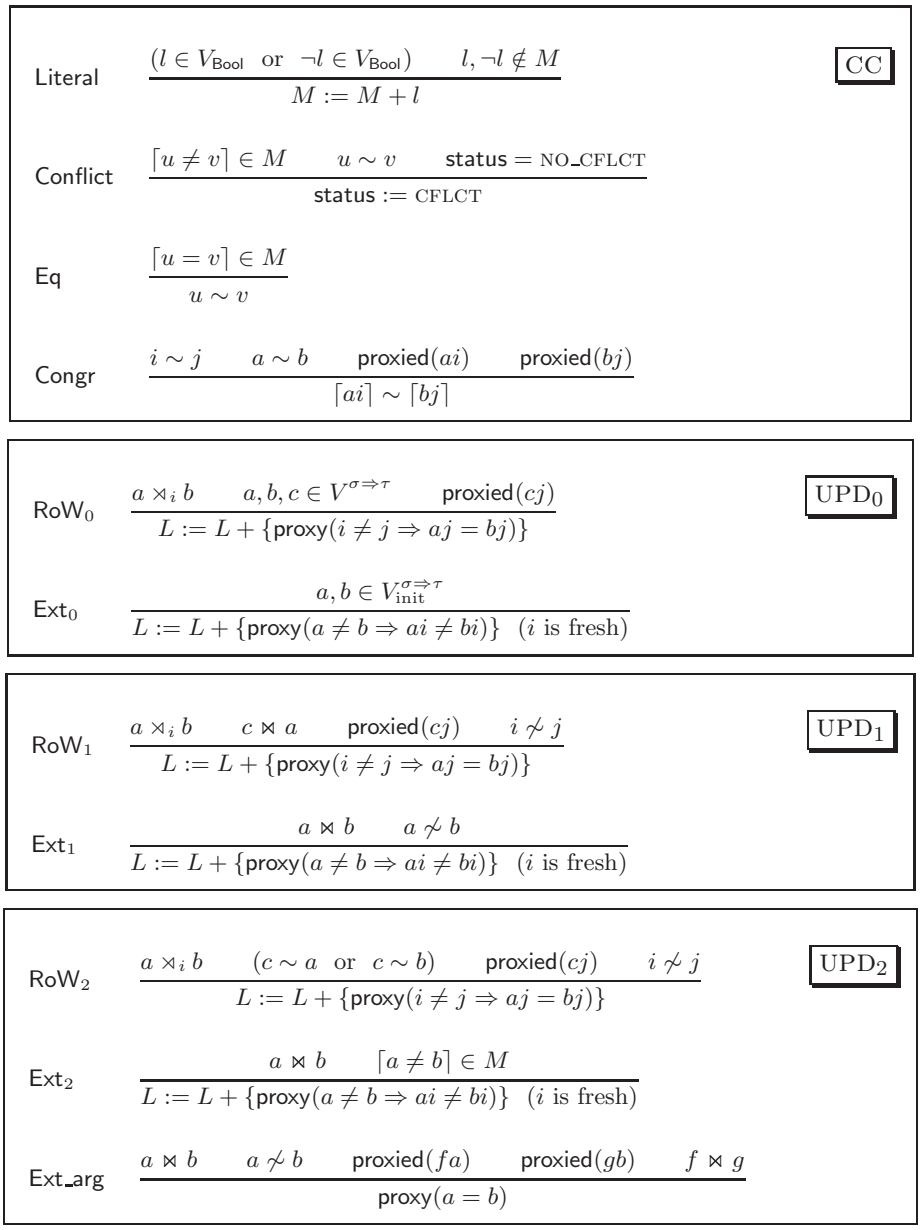

Fig. 2: Above and below the line in each rule are the rule's guard and action respectively. The guard is the enabling condition on the state, and the action is the state change. The rules in the top box describe a congruence-closure algorithm. These rules are part of systems UPD$_0$, UPD$_1$, UPD$_2$ as well. The lower three boxes present the read-over-write and extensionality rules that are specific for each of the three systems. The notation $V_{\text{init}}^{\sigma \Rightarrow \tau}$ in Ext$_0$ is for the initial set of variables of type $\sigma \Rightarrow \tau$, and $V^{\sigma \Rightarrow \tau}$ in RoW$_0$ is for the set of variables of type $\sigma \Rightarrow \tau$ that exist in the current state.

for any $a' \sim a$ and $b' \sim b$. Progress conditions are left implicit in Figure 2 in order to reduce clutter, but they are indispensable.

For a given input set Ψ_0 of proxy equations, the initial state $s_{\text{init}}(\Psi_0)$ is the tuple $\langle V, \Psi, [\,], \varnothing, \text{NO\_CFLCT}, \sim \rangle$, where

- Ψ is obtained by adding new proxy equations to Ψ_0 if necessary so that for every $b \triangleq U(a,i,x)$ in Ψ, we have $\mathsf{proxied}(bi)$ and—if x is of type Bool—$\mathsf{proxied}(ai)$ too[4]
- V is the set of all variables in Ψ
- $\sim$ is generated by $\lceil bi \rceil \sim x$, where $b \triangleq U(a,i,x)$ is in Ψ

It is straightforward to check that conditions **(i-iii)** are satisfied and so the systems $\text{UPD}_{0,1,2}$ are theory solvers in the sense of §2.2.

Theorem 1. UPD_0, UPD_1, *and* UPD_2 *are terminating, sound, and complete.*

We can offer here only some high-level remarks about the proof of Theorem 1.[5] The completeness part is the most difficult and, like other authors (e.g., [19, 7, 12]), we prove it by constructing a syntactic model for $\Lambda(s)$ when s is a final non-conflicting state. The amount of the "syntactic material" available in a final state directly affects the difficulty of the model construction. The system UPD_0 generously provides a syntactic witness for disequality of any two arrays a, b of the same type, as well as the information whether ai and bi are equal or not, for any index i. This information suffices to build a model. Indeed, when used with the strategy that applies Ext_0 exhaustively, then RoW_0 exhaustively, and then lets CC finish the job, UPD_0 is much like the reduction procedure of [12].

The conditions $c \bowtie a$ and $a \bowtie b$ in the guards of RoW_1 and Ext_1 greatly reduce the number of created lemmas. Intuitively, we do not lose completeness by imposing these restrictions because if two array variables are not in the same $\bowtie$-class, then we can rely on the (assumed) infinite cardinality of the index type to ensure a witness for their disequality. The additional guard constraints $i \not\sim j$ and $a \not\sim b$ in RoW_1 and Ext_1 are justified by the observation that the relation $\sim$ expresses what the system at a given state knows about implied equalities between variables. They further curtail the generation of lemmas and also suggest strategies that prioritize the use of CC rules over the lemma-generating ones.

In our final system UPD_2, creation of Ext-lemmas is not done until positively necessary: when an array disequality is explicitly asserted (put on the stack M). Unfortunately, this extremely frugal Ext_2 leads to an incomplete system: even if coupled with RoW_0, it would not refute the unsatisfiable query $\{i \neq j,\ b = U(a,i,x),\ b = U(a,j,y),\ fa \neq fb\}$ (example in §6.2 of [19]), because it would not generate the necessary extensionality lemma for $a = b$. A minimal remedy is provided by the rule $\mathsf{Ext\_arg}$; it will introduce a proxy variable for $a = b$ so that the SAT solver will eventually split on it, enabling the necessary firing of Ext_2.

[4] If we allowed concrete types other than Bool, we would need to forbid the finite ones to occur as index types, and we would need to impose this condition $\mathsf{proxied}(ai)$ whenever the type of x is finite.

[5] For the full proof, see `www.cs.uiowa.edu/~fuchs/PAPERS/array_solver.pdf`.

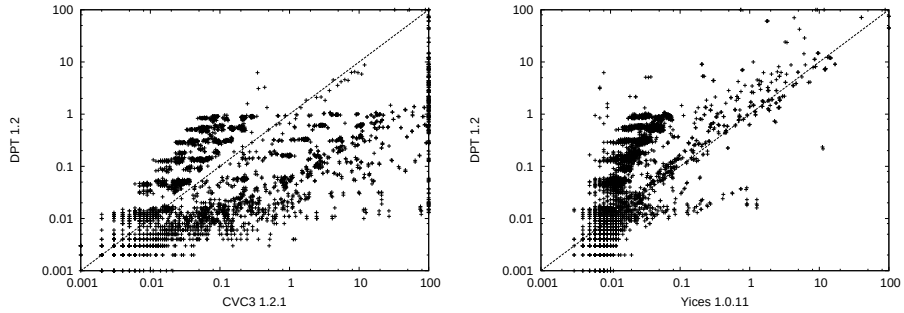

Fig. 3: *DPT* vs. *CVC3* and *Yices* on 3694 SMT-LIB benchmarks ("QF_AUFIDL").

4 Experiments

We have extended the congruence-closure module of the SMT solver *DPT*, written in *OCaml*, with the rules of the system UPD_2 [1]. Performance of our initial implementation[6] is compared in Figure 3 with *CVC3* [3] and *Yices* [10] on the set of QF_AUFLIA benchmarks whose arithmetical content is expressed in the difference logic [5]. (Of the three participants in the QF_AUFLIA category at the SMT-COMP 2007 competition, *Yices* was the winner, and *CVC3* was the only open-source solver.) *DPT* proved competitive with *Yices* and significantly better than *CVC3*. With 100 sec timeout, *DPT* used 2787 sec and timed out on 6 benchmarks, *CVC3* used 30729 sec and timed out on 230 benchmarks, and *Yices* used 1385 sec and timed out on 5 benchmarks.[7]

To emphasize the benefit of $\bowtie$-related frugality, we have also compared the three solvers on ten benchmarks $\Phi_{100}, \Phi_{200}, \ldots, \Phi_{1000}$, where Φ_n is the satisfiable formula $U((\cdots U(a_0, i_1, x_1), \ldots), i_n, x_n) \neq U((\cdots U(b_0, i_1, x_1), \ldots), i_n, x_n)$ expressing disequality of the results of the same sequence of n updates applied to a_0 and b_0. *DPT* takes less than a second even for Φ_{1000}, while *CVC3* times out (after 100 sec) on Φ_{400}, and *Yices* times out on Φ_{500}.

5 Conclusion

This paper is part of our ongoing project to design a high-performance SMT solver, open-sourced and with strong theoretical foundations [1, 14, 13]. We have given a rule-based axiom-instantiating solver for a *parametric* theory of arrays, where formulas can refer to multiple and arbitrarily nested array types. Our description lends itself to implementation in a fairly direct manner.

We have clarified the conditions that solvers of the axiom-instantiating and similar kinds must meet in order to correctly participate in an SMT combination

[6] We ran *DPT 1.2* with the `-q` option and the variable `OCAMLRUNPARAM` set to `51200000`.

[7] The performance on these benchmarks depends not only on the array decision procedure, but also on the linear arithmetic procedure, the combination method, preprocessing, and the implementation language.

framework, and we have proved that our array solver meets the conditions. Competitiveness of our implementation proves that we have struck a balance between frugality and completeness. In this, we have likely not achieved the optimum, but we have provided a setup for checking correctness of better algorithms to be found. We expect to use the same setup for building an axiom-instantiating solver for the basic theory of sets. We suspect there are other theories that could use specific frugal instantiation algorithms, perhaps the theory developed for verifying heap-manipulating programs in [15].

Acknowledgments. Thanks to Clark Barrett, Jim Grundy, Albert Rubio, and Cesare Tinelli with whom we discussed ideas about the array solver and received comments, criticism, or alternative solutions.

References

1. Decision Procedure Toolkit. `www.sourceforge.net/projects/DPT`, 2008.
2. A. Armando, S. Ranise, and M. Rusinowitch. A rewriting approach to satisfiability procedures. *Inf. Comput.*, 183(2):140–164, 2003.
3. C. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *Computer Aided Verification (CAV)*, vol. 3114 of *LNCS* , pp. 515–518. Springer, 2004.
4. C. Barrett, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Splitting on demand in SAT Modulo Theories. In *Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, vol. 4246 of *LNCS*, pp. 512–526. Springer, 2006.
5. C. Barrett, S. Ranise, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2008.
6. M. Bozzano et al. Efficient theory combination via boolean search. *Inf. Comput.*, 204(10):1493–1525, 2006.
7. A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In *Verification, Model Checking, and Abstract Interpretation: (VMCAI)*, vol. 3855 of *LNCS*, pp. 427–442. Springer, 2006.
8. R. Bryant, S. Lahiri, and S. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *Computer Aided Verification (CAV)*, vol. 2404 of *LNCS*, pp. 78–92. Springer, 2002.
9. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, 2005.
10. B. Dutertre and L. de Moura. The YICES SMT solver. Technical report, SRI International, 2006.
11. S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Decision procedures for extensions of the theory of arrays. *Annals of Mathematics and Artificial Intelligence*, 50(3-4):231–254, 2007.
12. D. Kapur and C. G. Zarba. A reduction approach to decision procedures. Technical Report TR-CS-1005-44, University of New Mexico, 2005.
13. S. Krstić and A. Goel. Architecting solvers for SAT Modulo Theories: Nelson-Oppen with DPLL. In *FroCoS*, vol. 4720 of *LNCS*, pp. 1–27. Springer, 2007.
14. S. Krstić, A. Goel, J. Grundy, and C. Tinelli. Combined satisfiability modulo parametric theories. In *TACAS*, vol. 4424 of *LNCS*, pp. 602–617. Springer, 2007.
15. S. Lahiri and S. Qadeer. Back to the future: revisiting precise program verification using SMT solvers. *SIGPLAN Not.*, 43(1):171–182, 2008.

16. G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.
17. R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: from an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, Nov. 2006.
18. R. Sebastiani. Lazy satisfiability modulo theories. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 3:141–224, 2007.
19. A. Stump, D. L. Dill, C. W. Barrett, and J. Levitt. A decision procedure for an extensional theory of arrays. In *Logic in Computer Science: 16th IEEE Symposium, LICS*, pp. 29–37. IEEE Press, 2001.

Pex - White Box Test Generation for .NET

Nikolai Tillmann

Microsoft Research, Redmond, USA
nikolait@microsoft.com

Abstract. Pex is a white-box test generation tool for .NET. Starting from a hand-written parameterized unit test, Pex analyzes the program-under-test to determine relevant test inputs fully automatically. To this end, Pex uses dynamic symbolic execution, where the program is executed multiple times with different inputs while the taken execution paths are monitored. An SMT-solver with model-generation capabilities computes new test inputs that will exercise different execution paths. The result is a traditional unit test suite with high code coverage. In one case study, we applied Pex to a core component of the .NET runtime which had already been extensively tested over several years. Pex found errors, including a serious issue.