The 2006 Federated Logic Conference

The Seattle Sheraton Hotel and Towers

Seattle, Washington

August 10 - 22, 2006

ICLP'06 Workshop

PDPAR'06: Pragmatical Aspects of Decision Procedures in Automated Reasoning

August 21st, 2006

Proceedings

Editors:

Byron Cook, Roberto Sebastiani

Table of Contents

Table of Contents iii
Program Committee iv
Additional Reviewers iv
Foreword v

Keynote contributions (abstracts)

Proof Procedures for Separated Heap Abstractions
Peter O'Hearn 1

The Power of Finite Model Finding
Koen Claessen 2

Original papers

Mothers of Pipelines
Krstic, Jones, O'Leary 3

Applications of hierarchical reasoning in the verification of complex systems
Jacobs, Sofronie-Stokkermans 15

Towards Automatic Proofs of Inequalities Involving Elementary Functions
Akbarpour, Paulson 27

Rewrite-Based Satisfiability Procedures for Recursive Data Structures
Bonacina, Echenim 38

An Abstract Decision Procedure for Satisfiability in the Theory of Recursive Data Types
Barrett, Shikanian, Tinelli 50

Presentation-only papers (abstracts)

A Framework for Decision Procedures in Program Verification
Strichman, Kroening 62

Easy Parameterized Verification of Biphase Mark and 8N1 Protocols
Brown, Pike 63

Predicate Learning and Selective Theory Deduction for Solving Difference Logic
Wang, Gupta, Ganai 64

Deciding Extensions of the Theory of Arrays by Integrating Decision Procedures and Instantiation Strategies
Ghilardi, Niccolini, Ranise, Zucchelli 65

Producing Conflict Sets for Combinations of Theories
Ranise, Ringeissen, Tran 66

Program Committee

Byron Cook (Microsoft Research, UK) [co-chair]
Roberto Sebastiani (Università di Trento, Italy) [co-chair]

Alessandro Armando (Università di Genova)
Clark Barrett (New York University)
Alessandro Cimatti (ITC-Irst, Trento)
Leonardo de Moura (SRI International)
Niklas Een (Cadence Design Systems)
Daniel Kroening (ETH-Zurich)
Shuvendu Lahiri (Microsoft Research)
Robert Nieuwenhuis (Technical University of Catalonia)
Silvio Ranise (LORIA, Nancy)
Eli Singerman (Intel Corporation)
Ofer Strichman (Technion)
Aaron Stump (Washington University)
Cesare Tinelli (University of Iowa)
Ashish Tiwari (Stanford Research Institute, SRI)

Additional reviewers

Nicolas Blanc
Juergen Giesl
Guillem Godoy
Kalyanasundaram Krishnamani
Michal Moskal
Enrica Nicolini
Albert Oliveras
Zvonimir Rakamaric
Simone Semprini
Armando Tacchella
Francesco Tapparo
Christoph Wintersteiger
Daniele Zucchelli

Foreword

This volume contains the proceedings of the 4th Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR'06), held in Seattle, USA, on August 21st, 2006, as part of the 2006 Federated Logic Conference (FLoC'06) and affiliated with the 3rd International Joint Conference on Automated Reasoning (IJCAR'06).

The applicative importance of decision procedures for the validity or the satisfiability problem in decidable first-order theories is being increasingly acknowledged in the verification community: many interesting and powerful decision procedures have been developed, and applied to the verification of word-level circuits, hybrid systems, pipelined microprocessors, and software.

The PDPAR'06 workshop has brought together researchers interested in both the theoretical and the pragmatical aspects of decision procedures, giving them a forum for presenting and discussing not only theoretical and algorithmic issues, but also implementation and evaluation techniques, with the ultimate goal of making new decision procedures possible and old decision procedures more powerful and more useful.

In this edition of PDPAR we have allowed not only original papers, but also "presentation-only papers", i.e., papers describing work previously published in non-FLOC'06 forums (which are not inserted in the proceedings). We are allowing the submission of previously published work in order to allow researchers to communicate good ideas that the PDPAR attendees are potentially unaware of.
The program included:

- two keynote presentations by Peter O'Hearn, University of London, and Koen Claessen, Chalmers University.
- 10 technical paper presentations, including 5 original papers and 5 "presentation-only" papers.
- A discussion session.

Additional details for PDPAR'06 (including the program) are available at the web site `http://www.dit.unitn.it/~rseba/pdpar06/`.

We gratefully acknowledge the financial support of Microsoft Research.

Seattle, August 2006

Byron Cook Microsoft Research, Cambridge, UK

Roberto Sebastiani DIT, University of Trento, Italy

Proof Procedures for Separated Heap Abstractions

(keynote presentation)

Peter O'Hearn
Queen Mary, University of London
ohearn@dcs.qmul.ac.uk

Abstract

Separation logic is a program logic geared towards reasoning about programs that mutate heap-allocated data structures. This talk describes ideas arising from joint work with Josh Berdine and Cristiano Calcagno on proof procedure for a sublogic of separation logic that is oriented to lightweight program verification and analysis. The proof theory uses ideas from substructural logic together with induction-free reasoning about inductive definitions of heap structures. Substructural reasoning is used to to infer frame axioms, which describe the portion of a heap that is not altered by a procedure, as well as to discharge verification conditions; more precisely, the leaves of failed proofs can give us candidate frame axioms. Full automation is achieved through the use of special axioms that capture properties that would normally be proven using by induction. I will illustrate the proof method through its use in the Smallfoot static assertion checker, where it is used to prove verification conditions and infer frame axioms, as well as in the Space Invader program analysis, where it is used to accelerate the convergence of fixed-point calculations.

The Power of Finite Model Finding

(keynote presentation)

Koen Claessen
Chalmers University of Technology and
Jasper Design Automation
koen@cs.chalmers.se

Abstract

Paradox is a tool that automatically finds finite models for first-order logic formulas, using incremental SAT. In this talk, I will present a new look on the problem of finding finite models for first-order logic formulas. In particular, I will present a novel application of finite model finding to the verification of finite and infinite state systems; here, a finite model finder can be used to automatically find abstractions of systems for use in safety property verification. In this verification process, it turns out to be vital to use typed (or sorted) first-order formulas. Finding models for typed formulas brings the freedom to use different domain sizes for each type. How to choose these different domain sizes is still very much an unexplored problem. We show how a simple extension to a SAT-solver can be used to guide the search for typed models with several domains of different sizes.

Mothers of Pipelines

Sava Krstić, Robert B. Jones, and John W. O'Leary

Strategic CAD Labs, Intel Corporation, Hillsboro, OR, USA

Abstract. We present a novel method for pipeline verification using SMT solvers. It is based on a non-deterministic "mother pipeline" machine (*MOP*) that abstracts the instruction set architecture (*ISA*). The *MOP* vs. *ISA* correctness theorem splits naturally into a large number of simple subgoals. This theorem reduces proving the correctness of a given pipelined implementation of the *ISA* to verifying that each of its transitions can be modeled as a sequence of *MOP* state transitions.

1 Introduction

Proving correctness of microarchitectural processor designs (*MA*) with respect to their instruction set architecture (*ISA*) amounts to establishing a simulation relation between the behaviors of *MA* and *ISA*. There are different ways in the literature to formulate the correctness theorem that relates the steps of the two machines [1], but the complexity of the *MA*'s step function remains the major impediment to practical verification. The challenge is to find a systematic way to break the verification effort into manageable pieces.

We propose a solution based on the obvious fact that the execution of any instruction can be seen as a sequence of smaller actions (let us call them *mini-steps* in this informal overview), and the observation that the mini-steps can be understood at an abstract level, without mentioning any concrete *MA*. Examples of mini-steps are fetching an instruction, getting an operand from the register file, having an operand forwarded by a previous instruction in the pipeline, writing a result to the register file, and retiring. We introduce an intermediate specification *MOP* between *ISA* and *MA* that describes the execution of each instruction as a sequence of mini-steps. By design, our highly non-deterministic intermediate specification admits a broad range of implementations. For example, *MOP* admits implementations that are out-of-order or not, speculative or not, superscalar or not, etc. This approach allows us to separate the implementation-independent proof obligations that relate *ISA* to *MOP* from those that rely upon the details of the *MA*. This can potentially amortize some of the proof effort over several different designs.

The concept of *parcels*, formalizing partially-executed instructions, will be needed for a thorough treatment of mini-steps. We will follow the intuition that from any given state of any *MA* one can always extract the current state of its *ISA* components and infer a queue of parcels currently present in the *MA* pipeline. In Section 2, we give a precise definition of a transition system *MOP* whose states are pairs of the form $\langle$*ISA* state, queue of parcels$\rangle$, and whose transitions are mini-steps as described above. Intuitively, it is clear that with a sufficiently complete set of mini-steps we will

be able to model any *MA* step in this transition system as a sequence of mini-steps. Similarly, it should be possible to express any *ISA* step as a sequence of mini-steps of *MOP*.

Figure 1 indicates that correctness of a microarchitecture *MA* with respect to *ISA* is implied by correctness results that relate these machines with *MOP*. In Section 3, we will prove the crucial *MOP* vs. *ISA* correctness property: despite its non-determinism, all *MOP* executions correspond to *ISA* executions. The proof rests on the local confluence of *MOP*. (Proofs are provided in the Appendix.)

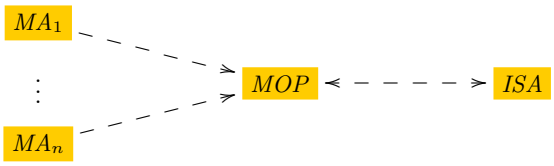

Fig. 1. With transitions that express atomic steps in instruction execution, a *mother of pipelines* *MOP* simulates the *ISA* and its multiple microarchitectural implementations. Simulation *à la* Burch-Dill flushing justifies the arrow from *MOP* to *ISA*.

The *MA* vs. *MOP* relationship is discussed in Section 4. We will see that all one needs to prove here is a precise form of the simulation mentioned above: there exists an abstraction function that maps *MA* states to *MOP* states such that for any two states joined by a *MA* transition, the corresponding *MOP* states are joined by a sequence of mini-steps.

MA vs. *MOP* vs. *ISA* correctness theorems systematically reduce to numerous subgoals, suitable for automated SMT solvers ("satisfiability modulo theories"). We used CVC Lite [4] and our initial experience is discussed in Section 5.

2 *MOP* Definition

The *MOP* definition depends on the *ISA* and the class of pipelined implementations that we are interested in. The particular *MOP* described in this section has a simple load-store *ISA* and can model complex superscalar implementations with out-of-order execution and speculation.

2.1 The Instruction Set Architecture

ISA is a deterministic transition system with system variables *pc* : IAddr, *rf* : RF, *mem* : MEM, *imem* : IMEM. We assume the types Reg and Word of registers and machine words, so that *rf* can be viewed as a Reg-indexed array with Word values. Similarly, *mem* can be viewed as a Word-indexed array with values in Word, while

instruction *imem.pc*	actions
opc1 dest src1 src2	$pc := pc + 4 \quad rf.dest := \mathsf{alu}\ opc1\ (rf.src1)\ (rf.src2)$
opc2 dest src1 imm	$pc := pc + 4 \quad rf.dest := \mathsf{alu}\ opc2\ (rf.src1)\ imm$
ld *dest src1 offset*	$pc := pc + 4 \quad rf.dest := mem.(rf.src1 + offset)$
st *src1 dest offset*	$pc := pc + 4 \quad mem.(rf.dest + offset) := rf.src1$
opc3 reg offset	$pc := \begin{cases} target \text{ if } taken \\ pc + 4 \text{ otherwise} \end{cases}$, where $target = get\_target\ pc\ offset$ $taken = get\_taken\ (get\_test\ opc3)\ (rf.reg)$

Fig. 2. *ISA* instruction classes (left column) and corresponding transitions. The variables $dest, src1, src2, reg$ have type Reg, and $imm, offset$ have type Word. For the three opcodes, we have $opc1 \in \{\mathsf{add}, \mathsf{sub}, \mathsf{mult}\}$, $opc2 \in \{\mathsf{addi}, \mathsf{subi}, \mathsf{multi}\}$, $opc3 \in \{\mathsf{beqz}, \mathsf{bnez}, \mathsf{j}\}$.

imem is an IAddr-indexed array with values in the type Instr of instructions. Instructions fall into five classes that are identified by the predicates *alu_reg*, *alu_imm*, *ld*, *st*, *branch*. The form of an instruction of each class is given in Figure 2. The figure also shows the *ISA* transitions—the change-of-state equations defined separately for each instruction class.

2.2 State

Parcels are records with the following fields:

$instr : \mathsf{Instr}_\perp$	$my\_pc : \mathsf{IAddr}_\perp$	$dest, src1, src2 : \mathsf{Reg}_\perp$
$imm : \mathsf{Word}_\perp$	$opc : \mathsf{Opcode}_\perp$	$data1, data2, res, mem\_addr : \mathsf{Word}_\perp$
$tkn : \mathsf{bool}_\perp$	$next\_pc : \mathsf{IAddr}_\perp$	$wb : \{\perp, \top\}$ $pc\_upd : \{\perp, \mathtt{s}, \mathtt{m}, \top\}$

The subscript $\perp$ to a type indicates the addition of the element $\perp$ ("undefined") to the type. The *empty_parcel* has all fields equal to $\perp$. The field *wb* indicates whether the parcel has written back to the register file (for arithmetical parcels and loads) or to the memory (for stores). Similiarly, *pc_upd* indicates whether the parcel has caused the update of *pc*. The additional values $\mathtt{s}$ and $\mathtt{m}$ are to record that the parcel has updated *pc* speculatively and that it mispredicted.

In addition to the architected state components *pc*, *rf*, *mem*, *imem*, the state of *MOP* contains integers *head* and *tail*, and a queue of parcels *q*. The queue is represented by an integer-indexed array with *head* and *tail* defining its front and back ends. We write *idx j* as an abbreviation for the predicate $head \le j \le tail$, saying that j is a valid index in q. The j^{th} parcel in q will be denoted $q.j$.

2.3 Transitions

The transitions of *MOP* are defined by the rules given in Figures 3 and 4. Each rule is a guarded parallel assignment described in the DEF/GRD/ACT format, where DEF contains local definitions, GRD (guard) is the set of predicates defining the rule's

domain, and ACT are the assignments made when the rule fires. Some rules contain additional predicates and functions, defined next.

The rule **decode** requires the predicate $\mathit{decoded}\ p \equiv p.\mathit{opc} \neq \bot$ and the function *decode* that updates the parcel field *opc* and some of the fields *dest*, *src1*, *src2*, *imm*. This update depends on the instruction class of $p.\mathit{instr}$, as in the following table.

instruction	*opc*	*dest*	*src1*	*src2*	*imm*
ADD R1 R2 R3	add	R1	R2	R3	$\bot$
ADDI R1 R2 17	addi	R1	R2	$\bot$	17
LD R1 R2 17	ld	R1	R2	$\bot$	17
ST R1 R2 17	st	$\bot$	R1	R2	17
BEQZ R1 17	beqz	$\bot$	R1	$\bot$	17
J 17	j	$\bot$	$\bot$	$\bot$	17

To specify how a given parcel should receive its *data1* and *data2*—from the register file or by forwarding—we use the predicates $\mathit{no\_mrw}\ r\ j \equiv (S = \emptyset)$ and $\mathit{mrw}\ r\ j\ k \equiv (S \neq \emptyset \wedge \max S = k)$, where $S = \{k \mid k < j \wedge \mathit{idx}\ k \wedge q.k.\mathit{dest} = r\}$. The former checks whether the parcel $q.j$ needs forwarding for a given register r and the latter gives the position k of the forwarding parcel (mrw = "*m*ost *r*ecent *w*rite").

The rule **write_back** allows parcels to write back to the register file out-of-order. The parcel $q.j$ can write back assuming (1) it is not mispredicted, and (2) there are no parcels in front of it that write to the same register or that have not fetched an operand from that register. These conditions are expressed by predicates $\mathit{fit}\ j \equiv \bigwedge_{\mathit{head}<j'\leq j} \mathit{fit\_at}\ j'$ and $\mathit{valid\_data\_upto}\ j \equiv \bigwedge_{\mathit{head}\leq j'\leq j} \mathit{valid\_data}\ j'$, where

$$\begin{aligned} \mathit{fit\_at}\ j &\equiv q.j.\mathit{my\_pc} = q.(j-1).\mathit{next\_pc} \neq \bot \\ \mathit{valid\_data}\ j &\equiv q.j.\mathit{data1} \neq \bot \wedge (\mathit{alu\_reg}\ q.j \Rightarrow q.j.\mathit{data2} \neq \bot) \end{aligned}$$

Memory access rules (**load** and **store**) enforce in-order execution of loads and stores. The existence and the location of the most recent memory access parcel are described by predicates *mrma* and *no_mrma*, analogous to *mrw* and *no_mrw* above: one has $\mathit{mrma}\ j\ k$ when k the largest valid index such that $k < j$ and $q.k$ is a load or store; and one has $\mathit{no\_mrma}\ j$ when no such number k exists. The completion of a parcel's memory access is formulated by

$$\mathit{ma\_complete}\ p \equiv (\mathit{load}\ p \wedge p.\mathit{res} \neq \bot) \vee (\mathit{store}\ p \wedge p.\mathit{wb} = \top).$$

The last four rules in Figure 3 cover the computation of the next pc value of a parcel, and the related test of whether the branch is taken and (if so) the computation of the target address. The functions *get_taken* and *get_target* are the same ones used by the *ISA*.

The rules **pc_update** and **speculate** govern the program counter updating. The first is based on the *next_pc* value of the last parcel and implements the regular *ISA* flow. The second implements practically unconstrained speculative updating of the pc, specified by an arbitrary branch_predict function.

Note that the status of a speculating branch changes when its *next_pc* value is computed; if the prediction is correct (matches *my_pc* of the next parcel), the change

DEF	$i = imem.pc$	**fetch**
GRD	$length = 0 \vee q.tail.pc\_upd \in \{\mathtt{s}, \top\}$	
ACT	$q.(tail + 1) := empty\_parcel[instr \mapsto i, my\_pc \mapsto pc] \quad tail := tail + 1$	
DEF	$p = q.j$	**decode** j
GRD	$idx\ j \quad \neg(decoded\ p)$	
ACT	$p := decode\ p$	
DEF	$p = q.j$	**data1_rf** j
GRD	$idx\ j \quad decoded\ p \quad p.src1 \neq \bot \quad p.data1 = \bot \quad no\_mrw\ (p.src1)\ j$	
ACT	$p.data1 := rf.(p.src1)$	
DEF	$p = q.j \quad \bar{p} = q.k,$ where $mrw\ (p.src1)\ j\ k$	**data1_forward** j
GRD	$idx\ j \quad decoded\ p \quad p.src1 \neq \bot \quad \bar{p}.res \neq \bot \quad p.data1 = \bot$	
ACT	$p.data1 := \bar{p}.res$	
DEF	$p = q.j \quad d = p.data1 \quad d' = \begin{cases} p.data2 & \text{if } alu\_reg\ p \\ p.imm & \text{if } alu\_imm\ p \end{cases}$	**result** j
GRD	$\begin{bmatrix} idx\ j \quad p.data1 \neq \bot \quad p.res = \bot \\ (alu\_reg\ p \wedge p.data2 \neq \bot) \vee alu\_imm\ p \end{bmatrix}$	
ACT	$p.res := alu\ p.opc\ \ d\ d'$	
DEF	$p = q.j \quad d = p.data1 \quad d' = p.data2$	**mem_addr** j
GRD	$idx\ j \quad p.mem\_addr = \bot \quad (ld\ p \wedge d \neq \bot) \vee (st\ p \wedge d' \neq \bot)$	
ACT	$p.mem\_addr := \begin{cases} d + p.imm & \text{if } ld\ p \\ d' + p.imm & \text{if } st\ p \end{cases}$	
DEF	$p = q.j$	**write_back** j
GRD	$\begin{bmatrix} idx\ j \quad alu\_reg\ p \vee alu\_imm\ p \vee ld\ p \quad fit\ j \quad valid\_data\_upto\ j \\ no\_mrw\ (p.dest)\ j \quad p.res \neq \bot \quad p.wb = \bot \end{bmatrix}$	
ACT	$rf.(p.dest) := p.res \quad p.wb := \top$	
DEF	$p = q.j$	**load** j
GRD	$\begin{bmatrix} idx\ j \quad ld\ p \quad p.mem\_addr \neq \bot \quad p.res = \bot \\ no\_mrma\ j \vee (mrma\ j\ k \wedge ma\_complete\ q.k) \end{bmatrix}$	
ACT	$p.res := mem.(p.mem\_addr)$	
DEF	$p = q.j$	**store** j
GRD	$\begin{bmatrix} idx\ j \quad st\ p \quad p.mem\_addr \neq \bot \quad p.data1 \neq \bot \quad p.wb = \bot \quad fit\ j \\ no\_mrma\ j \vee (mrma\ j\ k \wedge ma\_complete\ q.k) \end{bmatrix}$	
ACT	$mem.(p.mem\_addr) := p.data1 \quad p.wb := \top$	
DEF	$p = q.j$	**branch_target** j
GRD	$idx\ j \quad branch\ p \quad decoded\ p \quad p.res = \bot$	
ACT	$p.res := get\_target\ (p.my\_pc)\ (p.imm)$	
DEF	$p = q.j \quad t = get\_test\ (p.opc)$	**branch_taken** j
GRD	$idx\ j \quad branch\ p \quad decoded\ p \quad p.data1 \neq \bot \quad p.tkn = \bot$	
ACT	$p.tkn := get\_taken\ t\ (p.data1)$	
DEF	$p = q.j$	**next_pc_branch** j
GRD	$idx\ j \quad branch\ p \quad p.tkn \neq \bot \quad p.res \neq \bot \quad p.next\_pc = \bot$	
ACT	$p.next\_pc := \begin{cases} p.res & \text{if } p.tkn \\ (p.my\_pc) + 4 & \text{otherwise} \end{cases}$	
DEF	$p = q.j$	**next_pc_nonbranch** j
GRD	$idx\ j \quad \neg(branch\ p) \quad decoded\ p \quad p.next\_pc = \bot$	
ACT	$p.next\_pc := (p.my\_pc) + 4$	

Fig. 3. *MOP* transitions (Part 1). The rules **data2_rf** and **data2_forward** are analogous to **data1_rf** and **data1_forward**, and are not shown.

DEF	$p = q.tail$	**pc_update**
GRD	$length > 0 \quad decoded\ p \quad p.next\_pc \neq \bot \quad p.pc\_upd \neq \top$	
ACT	$pc := p.next\_pc \quad p.pc\_upd := \top$	
DEF	$p = q.tail$	**speculate**
GRD	$length > 0 \quad decoded\ p \quad branch\ p \quad p.pc\_upd = \bot \quad p.next\_pc = \bot$	
ACT	$pc :=$ branch_predict $p.my\_pc \quad p.pc\_upd := \mathtt{s}$	
DEF	$p = q.j$	**prediction_ok** j
GRD	$idx\ j \quad idx\ (j+1) \quad p.pc\_upd = \mathtt{s} \quad fit\_at\ (j+1)$	
ACT	$p.pc\_upd := \top$	
DEF	$p = q.j$	**squash** j
GRD	$idx\ j \quad idx\ (j+1) \quad p.pc\_upd = \mathtt{s} \quad \neg(fit\_at\ (j+1)) \quad p.next\_pc \neq \bot$	
ACT	$tail := j \quad p.pc\_upd := \mathtt{m}$	
DEF		**retire**
GRD	$length > 0 \quad complete\ (q.head)$	
ACT	$head := head + 1$	

Fig. 4. *MOP* transitions (Part 2)

is modeled by rule **prediction_ok**. And if the *next_pc* value turns out wrong, rule **squash** becomes enabled, effecting removal of all parcels following the mispredicting branch.

Rule **retire** fires only at parcels that have completed their expected modification of the architected state. *complete* p is defined by $(p.wb = \top) \wedge (p.pc\_upd = \top)$ for non-branches, and by $p.pc\_upd = \top$ for branches.

3 *MOP* Correctness

We call *MOP* states with empty queues *flushed* and considered them the initial states of the *MOP* transition system. The map $\gamma\colon s \longmapsto \langle s, empty\_queue \rangle$ establishes a bijection from *ISA* states to flushed *MOP* states.

Note that *MOP* simulates *ISA*: if s and s' are two consecutive *ISA* states, then there exists a sequence of *MOP* transitions that leads from $\gamma(s)$ to $\gamma(s')$. The sequence begins with **fetch** and proceeds depending on the class of the instruction that was fetched, keeping the queue size equal to one until the last transition **retire**. One can prove with little effort that a requisite sequence from $\gamma(s)$ to $\gamma(s')$ can always be found within the set described by the strategy

> **fetch** ; **decode** ; (**data1_rf** ‖ (**data1_rf** ; **data2_rf**)) ;
> (**result** ‖ **mem_addr** ‖ (**branch_taken** ; **branch_target**)) ; [**load** ‖ **store**] ;
> (**next_pc_branch** ‖ **next_pc_nonbranch**) ; **pc_update** ; **retire**

A *MOP invariant* is a property that holds for all states reachable from initial (flushed) states. Local confluence is *MOP*'s fundamental invariant.

Theorem 1. *Restricted to reachable states, MOP is locally confluent.*

We omit the proof of Theorem 1. Note, however, that proof of local confluence breaks down into lemmas—one for each pair of rules. For *MOP*, most of the cases are resolved by rule commutation: if $m_1 \xleftarrow{\rho_1} m \xrightarrow{\rho_2} m_2$ (i.e., ρ_i applies to the state m and leads from it to m_i), then $m_1 \xrightarrow{\rho_2} m' \xleftarrow{\rho_1} m_2$, for some m'. For the sake of illustration, we show in Figure 5 three examples when local confluence requires non-trivial resolution. Diagrams 1 and 2 show two ways of resolving the

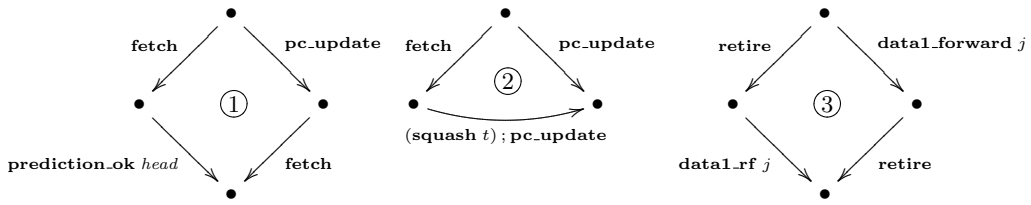

Fig. 5. Example non-trivial cases of local confluence

confluence of the rule pair (**fetch**, **pc_update**). Note that both rules are enabled only when $q.tail.pc\_upd = \mathtt{s}$. Thus, the parcel $q.tail$ must be a branch, and the fetch is speculative. Diagram 1 applies when the speculation goes wrong, Diagram 2 when the fetched parcel is correct. (In Diagram 2, t is the index of the branch at the tail of the original queue.) Diagram 3 shows local confluence for the pair (**retire**, **data1_forward** j) when $mrw\, j\, (q.j.src1)\, head$ holds.

The second fundamental property of *MOP* is related to termination. Even though *MOP* is not terminating (of course), every infinite run must have an infinite number of **fetch**es:

Lemma 1. *Without the rule* **fetch**, *MOP (on reachable states) is terminating and locally confluent.*

Proof. Every *MOP* rule except **fetch** either reduces the size of the queue, or makes a measurable progress in at least one of the fields of one parcel, while keeping all other fields the same. Measureable progress means going from $\bot$ to a non-$\bot$ value, or, in the case of the *pc_upd* field, going up in the ordering $\bot \prec \mathtt{s} \prec \mathtt{m} \prec \top$. This finishes the proof of termination. Local confluence of *MOP* without **fetch** follows from a simple analysis of the (omitted) proof of Theorem 1. □

Let us say that a *MOP* state is *irreducible* if none of the rules, except possibly **fetch** applies to it. It follows from Lemma 1, together with Newman's Lemma [3], that for every reachable state m there exists a unique irreducible state which can be reached from m using non-fetch rules. This state will be denoted $|m|$.

Lemma 2. *For every reachable state m, the irreducible state $|m|$ is flushed.*

Proof. Suppose the queue of $|m|$ is not empty and let p be its head parcel. We need to consider separately the cases defined by the instruction class of p. All cases being similar, we will give a proof only for one: when p is a conditional branch. Since **decode** does not apply to it, p must be fully decoded. Since **data1_rf** does not apply to p, we must have $p.data \neq \bot$ (other conditions in the guard of **data1_rf** are true). Now, since **branch_taken** and **branch_target** do not apply, we can conclude that $p.res \neq \bot$ and $p.tkn \neq \bot$. This, together with the fact that **next_pc_branch** does not apply, implies $p.next\_pc \neq \bot$. Now, if $p.pc\_upd = \top$, then **retire** would apply. Thus, we must have $p.pc\_upd \neq \top$. Since **pc_update** does not apply, the queue must have length at least 2. If $p.pc\_upd = \mathtt{s}$, then either **squash** or **prediction_ok** would apply to the parcel p. Thus, $p.pc\_upd$ is equal to $\bot$ or $\mathtt{m}$, and this contradicts the (easily checked) invariant saying that a parcel with $p.pc\_upd$ equal to $\bot$ or $\mathtt{m}$ must be at the tail of the queue. □

Define $\alpha(m)$ to be the *ISA* component of the flushed state $|m|$. Recall now the function γ defined at the beginning of this section. The functions γ and α map *ISA* states to *MOP* states and the other way around. Clearly, $\alpha(\gamma(s)) = s$.

The function α is analogous to the pipeline flushing functions of Burch-Dill [5]. Indeed, we can prove that *MOP* satisfies the fundamental Burch-Dill correctness property with respect to this flushing function.

Theorem 2. *Suppose a MOP transition leads from m to m', and m is reachable. Then $\alpha(m') = isa\_step\ (\alpha(m))$ or $\alpha(m') = \alpha(m)$.*

Proof. We can assume the transition $m \longrightarrow m'$ is a **fetch**; otherwise, we clearly have $|m| = |m'|$, and so $\alpha(m) = \alpha(m')$. The proof is by induction on the minimum length k of a chain of (non-fetch) transitions from m to $|m|$. If $k = 0$, then m is flushed, so $m = \gamma(s)$ for some *ISA* state s. By the discussion at the beginning of Section 3, the fetch transition $m \longrightarrow m'$ is the first in a sequence that, without using any further fetches, leads from $\gamma(s)$ to $\gamma(s')$, where $s' = isa\_step\ s$. It follows that $|m'| = |\gamma(s')|$, so $\alpha(m') = \alpha(\gamma(s')) = s'$, as required.

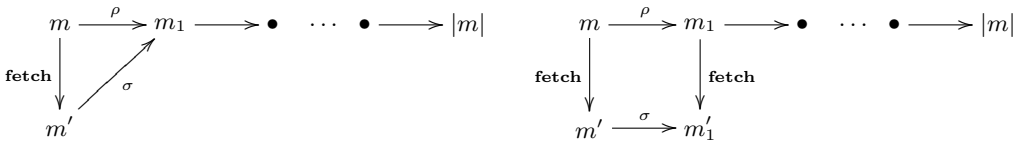

Fig. 6. Two cases for the inductive step in the proof of Theorem 2

Assume now $k > 0$ and let $m \xrightarrow{\rho} m_1$ be the first transition in a minimum length chain from m to $|m|$. Analyzing the proof of Theorem 1, one can see that local confluence in the case of the rule pair $(\mathbf{fetch}, \rho)$ can be resolved in one of the two ways shown in Figure 6, where σ has no occurrences of **fetch**. In the first case, we have $\alpha(m') = \alpha(m_1)$, and in the second case we have $\alpha(m') = \alpha(m'_1)$, where m'_1

is as in Figure 6. In the first case, we have $\alpha(m') = \alpha(m_1) = \alpha(m)$. In the second case, the proof follows from $\alpha(m) = \alpha(m_1)$, $\alpha(m') = \alpha(m'_1)$, and the induction hypothesis: $\alpha(m'_1) = \alpha(m_1)$ or $\alpha(m'_1) = \mathit{isa\_step}(\alpha(m'_1))$. □

4 Simulating Microarchitectures in *MOP*

Suppose *MA* is a microarchitecture purportedly implementing the *ISA*. We will say that a state-to-state map β from *MA* to *MOP* is a *MOP-simulation* if for every *MA* transition $s \longrightarrow_{MA} s'$, the state $\beta(s')$ is reachable in *MOP* from $\beta(s)$. Existence of a *MOP*-simulation proves (the safety part of) the correctness of *MA*. Indeed, for every execution sequence $s_1 \longrightarrow_{MA} s_2 \longrightarrow_{MA} \ldots$, we have $\beta(s_1) \longrightarrow^{+}_{MOP} \beta(s_2) \longrightarrow^{+}_{MOP} \ldots$, and then by Theorem 2, $\alpha(\beta(s_1)) \longrightarrow^{*}_{ISA} \alpha(\beta(s_2)) \longrightarrow^{*}_{ISA} \ldots$, demonstrating the crucial simulation relation between *MA* and *ISA*.

For a given *MA*, the *MOP*-simulation function β should be easy to guess. The difficult part is to verify that it satisfies the required property: the existence of a chain of *MOP* transitions $\beta(s) \longrightarrow^{+}_{MOP} \beta(s')$ for each transition $s \longrightarrow_{MA} s'$. Somewhat simplistically, this verification task can be partitioned as follows.

Suppose *MA*'s state variables are $v_1, \ldots, v_n$. (Among them are the *ISA* state variables, of course.) The *MA* transition function

$$s = \langle v_1, \ldots, v_n \rangle \longmapsto s' = \langle v'_1, \ldots, v'_n \rangle$$

is given by n functions $\mathit{next\_v}_i$ such that $v'_i = \mathit{next\_v}_i \langle v_1, \ldots, v_n \rangle$. The n-step sequence $s = s_0 \rightsquigarrow s_1 \rightsquigarrow \ldots \rightsquigarrow s_{n-1} \rightsquigarrow s_n = s'$ where $s_i = \langle v'_1, \ldots, v'_i, v_{i+1}, \ldots, v_n \rangle$ conveniently serializes the parallel computation that *MA* does when it makes a transition from s to s'. These n steps are not *MA* transitions themselves since the intermediate s_i need not be legitimate *MA* states at all. However, it is reasonable to expect that the progress described by this sequence is reflected in *MOP* by actual transitions:

$$\beta(s) = m_0 \longrightarrow^{*}_{MOP} m_1 \longrightarrow^{*}_{MOP} \ldots \longrightarrow^{*}_{MOP} m_n = \beta(s'). \tag{1}$$

Defining the intermediate *MOP* states m_i will usually be straightforward. Once they have been identified, the task of proving that $\beta(s')$ is reachable from $\beta(s)$ is broken down into n tasks of proving that m_{i+1} is reachable from m_i. Effectively, the correctness of the *MA* next-state function is reduced to proving a correctness property for each state component update function $\mathit{next\_v}_i$.

5 Mechanization

Our method is intended to be used with a combination of interactive (or manual) and automated theorem proving. The correctness of the *MOP* system (Theorem 2) rests largely on its local confluence (Theorem 1), which is naturally and easily split into a large number of cases that can be individually verified by an automated SMT

solver. The solver needs decision procedures for uninterpreted functions, a fragment of arithmetic, and common datatypes auch as arrays, records and enumeration types. Once the *MA*-simulation function β of Section 4 has been defined and the intermediate *MOP* states m_i in the chain (1) identified, it should also be possible to generate the proof of reachability of m_{i+1} from m_i with the aid of the same solver.

We have used manual proof decomposition and CVC Lite to implement the proof procedure just described. Our models for *ISA*, *MOP*, and *MA* are all written in the reflective general-purpose functional language *reFLect* [7]. In this convenient framework we can execute specifications and—through a HOL-like theorem prover on top of *reFLect* or an integrated CVC Lite—formally reason about them at the same time. Local confluence of *MOP* is (to some extent automatically) reduced to about 400 goals, which are individually proved with CVC Lite. For *MA* we used the textbook *DLX* model [9] and proved it is simulated in *MOP* by constructing the chains (1) and verifying them with CVC Lite. This proof is sketched in some detail in the Appendix.

Mechanization of our method is still in progress. Clean and efficient use of SMT solvers to prove properties of executable high-level models written in another language comes with challenges, some of which were discussed in [8]. For us, particularly exigent is the demand for heuristics for deciding when to expand occurrences of function symbols in goals passed to the SMT solver with the functions' definitions, and when to treat them as uninterpreted.

6 Related Work

The idea of flushing a pipeline automatically was introduced in a seminal paper by Burch and Dill [5]. In the original approach, all in-flight instructions in the implementation state are flushed out of the pipeline by inserting *bubbles*—NOPs that do not affect the program counter. Pipelines that use a combination of super-scalar execution, out-of-order execution, variable-latency execution units, etc. are too complex to flush directly. In response, researchers have invented a variety of ways, many based on flushing, to relate the implementation pipeline to the specification. We cover here only those approaches that are most closely related. The interested reader is refered to [1] for a relatively complete survey of pipeline verification approaches.

Damm and Pnueli [6] use a non-deterministic specification that generates all program traces that satisfy data dependencies. They use an intermediate abstraction with auxiliary variables to relate the specification and an implementation with out-of-order retirement based on Tomasulo's algorithm. In each step of the specification model, an entire instruction is executed atomically and its result written back. In the *MOP* approach, the execution of each instruction is broken into a sequence of mini-steps in order to relate to a pipelined implementation.

Skakkebæk *et al.* [16, 11] introduce *incremental flushing* and use a non-deterministic intermediate model to prove correctness of a simple out-of-order core with in-order retirement. Like us, they rely on arguments about re-ordering transactions.

While incremental flushing must deal with transactions as they are defined for the pipeline, we decompose pipeline transactions into much simpler "atomic" transactions. This facilitates a more general abstraction and should require significantly less manual proof effort than the incremental flushing approach.

Sawada and Hunt [14] use an intermediate model with an unbounded history table called a *micro-architectural execution trace table*. It contains instruction-specific information similar to that found in the *MOP* queue. Arons [2] follows a similar path, augmenting an implementation model with history variables that record the predicted results of instruction execution. In these approaches, auxiliary state is—like the *MOP* queue—employed to derive and prove invariants about the implementation's relation to the specification. While their auxiliary state is derived directly from the *MA*, *MOP* is largely independent of *MA* and has fine-grained transitions.

Arvind and Shen [15] use term rewriting to model an out-of-order processor and its specification. Similar to our approach, multiple rewrite rules may be required to complete an implementation step. As in the current approach, branch prediction is modeled by non-determinism. In contrast with the current approach, a single processor implementation is related directly to its in-order specification.

Hosabettu *et al.* [10] devised a method to decompose the Burch-Dill correctness statement into lemmas, one per pipeline stage. This inspired the decomposition we describe in Section 4.

Lahiri and Bryant [12], and Manolios and Srinivasan [13] verified complex microprocessor models using the SMT solver UCLID. Some *consistency invariants* in [12] occur naturally in our confluence proofs as well, but the overall approach is not closely related. The *WEB-refinement* method used in [13] produces proofs of strong correspondence between *ISA* and *MA* (stuttering bisimulation) that implies liveness. We have proved that *MOP* with a bounded queue and *ISA* are stuttering equivalent, also establishing liveness. The proof is contained in the Appendix.

7 Conclusion

We have presented a new approach for verifying a pipelined system $\mathcal{P}$ against its specification $\mathcal{S}$ by using an intermediate "pipeline mother" system $\mathcal{M}$ that explicates atomic computations occurring in steps of $\mathcal{S}$. For definiteness, we assumed that $\mathcal{P}$ is a microprocessor model and $\mathcal{S}$ is its *ISA*, but the method can potentially be applied to verify pipelined hardware components in general, or in protocol verification. This can all be seen as a refinement of the classical Burch-Dill method, but with the difficult flushing-based simulation pushed to the $\mathcal{M}$ vs. $\mathcal{S}$ level, where it amounts to proving local confluence of $\mathcal{M}$—a conjunction of easily-stated properties of limited size, readily verifiable by SMT solvers.

As an example, we specified a concrete intermediate model *MOP* for a simple load-store architecture and proved its correctness. We also verified the textbook machine *DLX* against it. However, our *MOP* contains more than is needed for verifying *DLX*: it is designed for simulation of microprocessor models with complex

out-of-order execution that cannot be handled by currently available methods. This will be addressed in future work. Also left for future work are improvements to our methodology (manual decomposition of verification goals into subgoals which we prove with CVC Lite [4]) and performance comparison with other published methods.

Acknowledgment. We thank Jesse Bingham for his comments.

References

1. M. D. Aagaard, B. Cook, N. A. Day, and R. B. Jones. A framework for superscalar microprocessor correctness statements. *Software Tools for Technology Transfer*, 4(3):298–312, 2003.
2. T. Arons. Verification of an advanced MIPS-type out-of-order execution algorithm. In *Computer Aided Verification (CAV'04)*, volume 3114 of *LNCS*, pages 414–426, 2004.
3. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
4. C. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. In R. Alur and D. A. Peled, editors, *Computer Aided Verification (CAV'04)*, volume 3114 of *LNCS*, pages 515–518, 2004.
5. J. Burch and D. Dill. Automatic verification of pipelined microprocessor control. In D. L. Dill, editor, *Computer Aided Verification (CAV'94)*, volume 818 of *LNCS*, pages 68–80, 1994.
6. W. Damm and A. Pnueli. Verifying out-of-order executions. In H. F. Li and D. K. Probst, editors, *Correct Hardware Design and Verification Methods (CHARME'97)*, pages 23–47. Chapman and Hall, 1997.
7. J. Grundy, T. Melham, and J. O'Leary. A reflective functional language for hardware design and theorem proving. *J. Functional Programming*, 16(2):157–196, 2006.
8. J. Grundy, T. F. Melham, S. Krstić, and S. McLaughlin. Tool building requirements for an API to first-order solvers. *Electr. Notes Theor. Comput. Sci.*, 144(2):15–26, 2006.
9. J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 1995.
10. R. Hosabettu, G. Gopalakrishnan, and M. Srivas. Verifying advanced microarchitectures that support speculation and exceptions. In E. A. Emerson and A. P. Sistla, editors, *Computer Aided Verification (CAV'00)*, volume 1855 of *LNCS*, pages 521–537, 2000.
11. R. B. Jones. *Symbolic Simulation Methods for Industrial Formal Verification.* Kluwer, 2002.
12. S. K. Lahiri and R. E. Bryant. Deductive verification of advanced out-of-order microprocessors. In W. A. Hunt Jr. and F. Somenzi, editors, *Computer Aided Verification (CAV'03)*, volume 2725 of *LNCS*, pages 341–354, 2003.
13. P. Manolios and S. K. Srinivasan. A complete compositional reasoning framework for the efficient verification of pipelined machines. In *IEEE/ACM International conference on Computer-aided design (ICCAD'05)*, pages 863–870. IEEE Computer Society, 2005.
14. J. Sawada and W. Hunt. Processor verification with precise exceptions and speculative execution. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer Aided Verification (CAV'98)*, volume 1427 of *LNCS*, pages 135–146, 1998.
15. X. Shen and Arvind. Design and verification of speculative processors. In *Workshop on Formal Techniques for Hardware*, Maarstrand, Sweden, June 1998.
16. J. Skakkebæk, R. Jones, and D. Dill. Formal verification of out-of-order execution using incremental flushing. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer Aided Verification (CAV'98)*, volume 1427 of *LNCS*, pages 98–109, 1998.

Applications of hierarchical reasoning in the verification of complex systems

Swen Jacobs and Viorica Sofronie-Stokkermans

Max-Planck-Institut für Informatik, Stuhlsatzenhausweg 85, Saarbrücken, Germany
e-mail: {sjacobs, sofronie}@mpi-sb.mpg.de

Abstract

In this paper we show how hierarchical reasoning can be used to verify properties of complex systems. Chains of local theory extensions are used to model a case study taken from the European Train Control System (ETCS) standard, but considerably simplified. We show how testing invariants and bounded model checking can automatically be reduced to checking satisfiability of ground formulae over a base theory.

1 Introduction

Many problems in computer science can be reduced to proving satisfiability of conjunctions of (ground) literals modulo a background theory. This theory can be a standard theory, the extension of a base theory with additional functions (free or subject to additional conditions), or a combination of theories. In [8] we showed that for special types of theory extensions, which we called *local*, hierarchic reasoning in which a theorem prover for the base theory is used as a "black box" is possible. Many theories important for computer science are local extensions of a base theory. Several examples (including theories of data structures, e.g. theories of lists (or arrays cf. [3]); but also theories of monotone functions or of functions satisfying semi-Galois conditions) are given in [8] and [9]. Here we present additional examples of local theory extensions occurring in the verification of complex systems.

In this paper we address a case study taken from the specification of the European Train Control System (ETCS) standard [2], but considerably simplified, namely an example of a communication device responsible for a given segment of the rail track, where trains may enter and leave. We suppose that, at fixed moments in time, all knowledge about the current positions of the trains is available to a controller which accordingly imposes constraints on the speed of some trains, or allows them to move freely within the allowed speed range on the track. Related problems were tackled before with methods from verification [2].

The approach we use in this paper is different from previously used methods. We use sorted arrays (or monotonely decreasing functions) for storing the train positions. The use of abstract data structures allows us to

pass in an elegant way from verification of several finite instances of problems (modeled by finite-state systems) to general verification results, in which sets of states are represented using formulae in first-order logic, by keeping the number of trains as a parameter. We show that for invariant or bounded model checking the specific properties of "position updates" can be expressed in a natural way by using chains of local theory extensions. Therefore we can use results in hierarchic theorem proving both for invariant and for bounded model checking[1]. By using locality of theory extensions we also obtained formal arguments on possibilities of systematic "slicing" (for bounded model checking): we show that for proving (disproving) the violation of the safety condition we only need to consider those trains which are in a 'neighborhood' of the trains which violate the safety condition[2].

Structure of the paper. Section 2 contains the main theoretical results needed in the paper. In Section 3 we describe the case study we consider. In Section 4 we present a method for invariant and bounded model checking based on hierarchical reasoning. Section 5 contains conclusions and perspectives.

2 Preliminaries

Theories and models. Theories can be regarded as sets of formulae or as sets of models. Let $\mathcal{T}$ be a theory in a (many-sorted) signature $\Pi = (S, \Sigma, \mathsf{Pred})$, where S is a set of sorts, Σ is a set of function symbols and Pred a set of predicate symbols (with given arities). A Π-structure is a tuple

$$\mathcal{M} = (\{M_s\}_{s\in S}, \{f_{\mathcal{M}}\}_{f\in\Sigma}, \{P_{\mathcal{M}}\}_{P\in\mathsf{Pred}}),$$

where for every $s \in S$, M_s is a non-empty set, for all $f \in \Sigma$ with arity $a(f)=s_1 \ldots s_n \rightarrow s$, $f_{\mathcal{M}} : \prod_{i=1}^{n} M_{s_i} \rightarrow M_s$ and for all $P \in \mathsf{Pred}$ with arity $a(P) = s_1 \ldots s_n, P_{\mathcal{M}} \subseteq M_{s_1} \times \ldots \times M_{s_n}$. We consider formulae over variables in a (many-sorted) family $X = \{X_s \mid s \in S\}$, where for every $s \in S$, X_s is a set of variables of sort s. A model of $\mathcal{T}$ is a Π-structure satisfying all formulae of $\mathcal{T}$. In this paper, whenever we speak about a theory $\mathcal{T}$ we implicitly refer to the set $\mathsf{Mod}(\mathcal{T})$ of all models of $\mathcal{T}$, if not otherwise specified.

Partial structures. Let $\mathcal{T}_0$ be a theory with signature $\Pi_0 = (S_0, \Sigma_0, \mathsf{Pred})$. We consider extensions $\mathcal{T}_1$ of $\mathcal{T}_0$ with signature $\Pi = (S, \Sigma, \mathsf{Pred})$, where $S = S_0 \cup S_1, \Sigma = \Sigma_0 \cup \Sigma_1$ (i.e. the signature is extended by new sorts and function symbols) and $\mathcal{T}_1$ is obtained from $\mathcal{T}_0$ by adding a set $\mathcal{K}$ of (universally quantified) clauses. Thus, $\mathsf{Mod}(\mathcal{T}_1)$ consists of all Π-structures which are models of $\mathcal{K}$ and whose reduct to Π_0 is a model of $\mathcal{T}_0$.

A *partial* Π*-structure* is a structure $\mathcal{M} = (\{M_s\}_{s\in S}, \{f_{\mathcal{M}}\}_{f\in\Sigma}, \{P_{\mathcal{M}}\}_{P\in\mathsf{Pred}})$, where for every $s \in S$, M_s is a non-empty set and for every $f \in \Sigma$ with arity

[1] Here we only focus on one example. However, we also used this technique for other case studies (among which one is mentioned – in a slightly different context – in [9]).

[2] In fact, it turns out that slicing (locality) results with a similar flavor presented by Necula and McPeak in [6] have a similar theoretical justification.

$s_1 \dots s_n \to s$, $f_{\mathcal{M}}$ is a partial function from $M_{s_1} \times \cdots \times M_{s_n}$ to M_s. The notion of evaluating a term t with variables $X = \{X_s \mid s \in S\}$ w.r.t. an assignment $\{\beta_s{:}X_s \to M_s \mid s \in S\}$ for its variables in a partial structure $\mathcal{M}$ is the same as for total many-sorted algebras, except that this evaluation is undefined if $t = f(t_1, \dots, t_n)$ with $a(f) = (s_1 \dots s_n \to s)$, and at least one of $\beta_{s_i}(t_i)$ is undefined, or else $(\beta_{s_1}(t_1), \dots, \beta_{s_n}(t_n))$ is not in the domain of $f_{\mathcal{M}}$. In what follows we will denote a many-sorted variable assignment $\{\beta_s{:}X_s \to M_s \mid s \in S\}$ as $\beta : X \to \mathcal{M}$. Let $\mathcal{M}$ be a partial Π-structure, C a clause and $\beta : X \to \mathcal{M}$. We say that $(\mathcal{M}, \beta) \models_w C$ iff either (i) for some term t in C, $\beta(t)$ is undefined, or else (ii) $\beta(t)$ is defined for all terms t of C, and there exists a literal L in C s.t. $\beta(L)$ is true in $\mathcal{M}$. $\mathcal{M}$ *weakly satisfies* C (notation: $\mathcal{M} \models_w C$) if $(\mathcal{M}, \beta) \models_w C$ for all assignments β. $\mathcal{M}$ *weakly satisfies (is a weak partial model of) a set of clauses* $\mathcal{K}$ (notation: $\mathcal{M} \models_w \mathcal{K}$, $\mathcal{M}$ is a w.p.model of $\mathcal{K}$) if $\mathcal{M} \models_w C$ for all $C \in \mathcal{K}$.

Local theory extensions. Let $\mathcal{K}$ be a set of (universally quantified) clauses in the signature $\Pi = (S, \Sigma, \mathsf{Pred})$, where $S = S_0 \cup S_1$ and $\Sigma = \Sigma_0 \cup \Sigma_1$. In what follows, when referring to sets G of ground clauses we assume they are in the signature $\Pi^c = (S, \Sigma \cup \Sigma_c, \mathsf{Pred})$ where Σ_c is a set of new constants. An extension $\mathcal{T}_0 \subseteq \mathcal{T}_0 \cup \mathcal{K}$ is *local* if satisfiability of a set G of clauses with respect to $\mathcal{T}_0 \cup \mathcal{K}$, only depends on $\mathcal{T}_0$ and those instances $\mathcal{K}[G]$ of $\mathcal{K}$ in which the terms starting with extension functions are in the set $\mathsf{st}(\mathcal{K}, G)$ of ground terms which already occur in G or $\mathcal{K}$. Formally,

$$\begin{aligned}\mathcal{K}[G] = \{C\sigma \mid & C \in \mathcal{K}, \text{ for each subterm } f(t) \text{ of } C, \text{ with } f \in \Sigma_1, \\ & f(t)\sigma \in \mathsf{st}(\mathcal{K}, G), \text{ and for each variable } x \text{ which does not} \\ & \text{occur below a function symbol in } \Sigma_1, \sigma(x) = x\},\end{aligned}$$

and $\mathcal{T}_0 \subseteq \mathcal{T}_1{=}\mathcal{T}_0 \cup \mathcal{K}$ is a local extension if it satisfies condition (Loc):

(Loc) For every set G of ground clauses $G \models_{\mathcal{T}_1} \bot$ iff there is no partial Π^c-structure P such that $P_{|\Pi_0}$ is a total model of $\mathcal{T}_0$, all terms in $\mathsf{st}(\mathcal{K}, G)$ are defined in P, and P weakly satisfies $\mathcal{K}[G] \wedge G$.

In [8, 9] we gave several examples of local theory extensions: e.g. any extension of a theory with free function symbols; extensions with selector functions for a constructor which is injective in the base theory; extensions of several partially ordered theories with monotone functions. In Section 4.2 we give additional examples which have particular relevance in verification.

Hierarchic reasoning in local theory extensions. Let $\mathcal{T}_0 \subseteq \mathcal{T}_1{=}\mathcal{T}_0 \cup \mathcal{K}$ be a local theory extension. To check the satisfiability of a set G of ground clauses w.r.t. $\mathcal{T}_1$ we can proceed as follows (for details cf. [8]):

Step 1: Use locality. By the locality condition, G is unsatisfiable w.r.t. $\mathcal{T}_1$ iff $\mathcal{K}[G] \wedge G$ has no weak partial model in which all the subterms of $\mathcal{K}[G] \wedge G$ are defined, and whose restriction to Π_0 is a total model of $\mathcal{T}_0$.

Step 2: Flattening and purification. We purify and flatten $\mathcal{K}[G] \wedge G$ by introducing new constants for the arguments of the extension functions as well as for the (sub)terms $t = f(g_1, \ldots, g_n)$ starting with extension functions $f \in \Sigma_1$, together with new corresponding definitions $c_t \approx t$. The set of clauses thus obtained has the form $\mathcal{K}_0 \wedge G_0 \wedge D$, where D is a set of ground unit clauses of the form $f(c_1, \ldots, c_n) \approx c$, where $f \in \Sigma_1$ and $c_1, \ldots, c_n, c$ are constants, and $\mathcal{K}_0, G_0$ are clause sets without function symbols in Σ_1.

Step 3: Reduction to testing satisfiability in $\mathcal{T}_0$. We reduce the problem to testing satisfiability in $\mathcal{T}_0$ by replacing D with the following set of clauses:

$$N_0 \quad = \quad \bigwedge\{\bigwedge_{i=1}^{n} c_i \approx d_i \rightarrow c \approx d \mid f(c_1, \ldots, c_n) = c, f(d_1, \ldots, d_n) = d \in D\}.$$

Theorem 1 ([8]) *With the notations above, the following are equivalent:*

(1) $\mathcal{T}_0 \wedge \mathcal{K} \wedge G$ has a model.

(2) $\mathcal{T}_0 \wedge \mathcal{K}[G] \wedge G$ has a w.p.model (where all terms in $\mathsf{st}(\mathcal{K}, \mathsf{G})$ are defined).

(3) $\mathcal{T}_0 \wedge \mathcal{K}_0 \wedge G_0 \wedge D$ has a w.p.model (with all terms in $\mathsf{st}(\mathcal{K}, \mathsf{G})$ defined).

(4) $\mathcal{T}_0 \wedge \mathcal{K}_0 \wedge G_0 \wedge N_0$ has a (total) Σ_0-model.

3 The RBC Case Study

The case study we discuss here is taken from the specification of the European Train Control System (ETCS) standard: we consider a radio block center (RBC), which communicates with all trains on a given track segment. Trains may enter and leave the area, given that a certain maximum number of trains on the track is not exceeded. Every train reports its position to the RBC in given time intervals and the RBC communicates to every train how far it can safely move, based on the position of the preceding train. It is then the responsibility of the trains to adjust their speed between given minimum and maximum speeds.

For a first try at verifying properties of this system, we have considerably simplified it: we abstract from the communication issues in that we always evaluate the system after a certain time $\Delta\mathsf{t}$, and at these evaluation points the positions of all trains are known. Depending on these positions, the possible speed of every train until the next evaluation is decided: if the distance to the preceding train is less than a certain limit $\mathsf{l}_{\mathsf{alarm}}$, the train may only move with minimum speed min (otherwise with any speed between min and the maximum speed max).

3.1 Formal Description of the System Model

We present two formal system models. In the first one we have a fixed number of trains; in the second we allow for entering and leaving trains.

Model 1: Fixed Number of Trains. In this simpler model, any state of the system is characterized by the following functions and constants:

- $\Delta\mathsf{t} > 0$, the time between evaluations of the system.
- min and max, the minimum and maximum speed of trains. We assume that $0 \leq \mathsf{min} \leq \mathsf{max}$.
- $\mathsf{l_{alarm}}$, the distance between trains which is deemed secure.
- n, the number of trains.
- pos, a function which maps indices (between 0 and $\mathsf{n}-1$) associated to trains on the track to the positions of those trains on the track. Here $\mathsf{pos}(i)$ denotes the current position of the train with index i.

We use a new function pos' to model the evolution of the system: $\mathsf{pos}'(i)$ denotes the position of i at the next evaluation point (after $\Delta\mathsf{t}$ time units). The way positions change (i.e. the relationship between pos and pos') is defined by the following set $\mathcal{K}_f = \{\mathsf{F1}, \mathsf{F2}, \mathsf{F3}, \mathsf{F4}\}$ of axioms:

(F1) $\quad \forall i \quad (i = 0 \quad \rightarrow \quad \mathsf{pos}(i) + \Delta\mathsf{t}{*}\mathsf{min} \leq_{\mathbb{R}} \mathsf{pos}'(i) \leq_{\mathbb{R}} \mathsf{pos}(i) + \Delta\mathsf{t}{*}\mathsf{max})$

(F2) $\quad \forall i \quad (0 < i < \mathsf{n} \quad \wedge \quad \mathsf{pos}(i-1) >_{\mathbb{R}} 0 \quad \wedge \quad \mathsf{pos}(p(i)) - \mathsf{pos}(i) \geq_{\mathbb{R}} \mathsf{l_{alarm}}$
$\rightarrow \mathsf{pos}(i) + \Delta\mathsf{t} * \mathsf{min} \leq_{\mathbb{R}} \mathsf{pos}'(i) \leq_{\mathbb{R}} \mathsf{pos}(i) + \Delta\mathsf{t}{*}\mathsf{max})$

(F3) $\quad \forall i \quad (0 < i < \mathsf{n} \quad \wedge \quad \mathsf{pos}(i-1) >_{\mathbb{R}} 0 \quad \wedge \quad \mathsf{pos}(p(i)) - \mathsf{pos}(i) <_{\mathbb{R}} \mathsf{l_{alarm}}$
$\rightarrow \mathsf{pos}'(i) = \mathsf{pos}(i) + \Delta\mathsf{t}{*}\mathsf{min})$

(F4) $\quad \forall i \quad (0 < i < \mathsf{n} \quad \wedge \quad \mathsf{pos}(i-1) \leq_{\mathbb{R}} 0 \quad \rightarrow \quad \mathsf{pos}'(i) = \mathsf{pos}(i)),$

Note that the train with number 0 is the train with the greatest position, i.e. we count trains from highest to lowest position.

Axiom F1 states that the first train may always move at any speed between min and max. F2 states that the other trains can do so if their predecessor has already started and the distance to it is larger than $\mathsf{l_{alarm}}$. If the predecessor of a train has started, but is less than $\mathsf{l_{alarm}}$ away, then the train may only move at speed min (axiom F3). F4 requires that a train may not move at all if its predecessor has not started.

Model 2: Incoming and leaving trains. If we allow incoming and leaving trains, we additionally need a measure for the number of trains on the track. This is given by additional constants first and last, which at any time give the number of the first and last train on the track (again, the first train is supposed to be the train with the highest position). Furthermore, the maximum number of trains that is allowed to be on the track simultaneously is given by a constant $\mathsf{maxTrains}$. These three values replace the number of trains n in the simpler model, the rest of it remains the same except that the function pos is now defined for values between first and last, where before it was defined between 0 and $\mathsf{n}-1$. The behavior of this extended system is described by the following set $\mathcal{K}_v$ consisting of axioms (V1) – (V9):

(V1) $\quad \forall i \quad (i = \mathsf{first} \rightarrow \mathsf{pos}(i) + \Delta\mathsf{t} * \mathsf{min} \leq_{\mathbb{R}} \mathsf{pos}'(i) \leq_{\mathbb{R}} \mathsf{pos}(i) + \Delta\mathsf{t} * \mathsf{max})$

(V2) $\quad \forall i \quad (\mathsf{first} < i \leq \mathsf{last} \wedge \mathsf{pos}(i-1) >_{\mathbb{R}} 0 \wedge \mathsf{pos}(i-1) - \mathsf{pos}(i) \geq_{\mathbb{R}} \mathsf{l_{alarm}}$
$\rightarrow \mathsf{pos}(i) + \Delta\mathsf{t} * \mathsf{min} \leq_{\mathbb{R}} \mathsf{pos}'(i) \leq_{\mathbb{R}} \mathsf{pos}(i) + \Delta\mathsf{t} * \mathsf{max})$

$$\text{(V3)} \quad \forall i \quad (\mathsf{first} < i \leq \mathsf{last} \wedge \mathsf{pos}(i-1) >_{\mathbb{R}} 0 \wedge \mathsf{pos}(i-1) - \mathsf{pos}(i) <_{\mathbb{R}} \mathsf{l_{alarm}} \rightarrow \mathsf{pos}'(i) = \mathsf{pos}(i) + \Delta\mathsf{t} * \mathsf{min})$$

$$\text{(V4)} \quad \forall i \quad (\mathsf{first} < i \leq \mathsf{last} \wedge \mathsf{pos}(i-1) \leq_{\mathbb{R}} 0 \rightarrow \mathsf{pos}'(i) = \mathsf{pos}(i))$$

$$\text{(V5)} \quad \mathsf{last} - \mathsf{first} + 1 < \mathsf{maxTrains} \rightarrow \mathsf{last}' = \mathsf{last} \vee \mathsf{last}' = \mathsf{last} + 1$$

$$\text{(V6)} \quad \mathsf{last} - \mathsf{first} + 1 = \mathsf{maxTrains} \rightarrow \mathsf{last}' = \mathsf{last}$$

$$\text{(V7)} \quad \mathsf{last} - \mathsf{first} + 1 > 0 \rightarrow \mathsf{first}' = \mathsf{first} \vee \mathsf{first}' = \mathsf{first} + 1$$

$$\text{(V8)} \quad \mathsf{last} - \mathsf{first} + 1 = 0 \rightarrow \mathsf{first}' = \mathsf{first},$$

$$\text{(V9)} \quad \mathsf{last}' = \mathsf{last} + 1 \quad \rightarrow \quad \mathsf{pos}'(\mathsf{last}') <_{\mathbb{R}} \mathsf{pos}'(\mathsf{last})$$

where primed symbols denote the state of the system at the next evaluation.

Here, axioms V1 − V4 are similar to F1 − F4, except that the fixed bounds are replaced by the constants first and last. V5 states that if the number of trains is less than maxTrains, then a new train may enter or not. V6 says that no train may enter if maxTrains is already reached. V7 and V8 are similar conditions for leaving trains. Finally, V9 states that if a train enters, its position must be behind the train that was last before.

4 Hierarchical reasoning in verification

The safety condition which is important for this type of systems is collision freeness. Intuitively (but in a very simplified model of the system of trains) collision freeness is similar to a 'bounded strict monotonicity' property for the function pos which stores the positions of the trains:

$$\mathsf{Mon}(\mathsf{pos}) \qquad \forall i, j \qquad (0 \leq i < j < \mathsf{n} \rightarrow \mathsf{pos}(i) >_{\mathbb{R}} \mathsf{pos}(j))$$

Mon(pos) expresses the condition that for all trains i, j on the track, if i precedes j then i should be positioned strictly ahead of j.

We will also consider a more realistic extension, which allows to express collision-freeness when the maximum length of the trains is known. In both cases, we focus on invariant checking and on bounded model checking.

4.1 Problems: Invariant checking, bounded model checking

In what follows we illustrate the ideas for the simple approach, in which collision-freeness is identified with strict monotonicity of the function which stores the positions of the trains. To check that strict monotonicity of train positions is an invariant, we need to check that:

(a) In the initial state the train positions (expressed by a function pos_0) satisfy the strict monotonicity condition $\mathsf{Mon}(\mathsf{pos}_0)$.

(b) Assuming that at a given state, the function pos (indicating the positions) satisfies the strict monotonicity condition Mon(pos), and the next state positions, stored in pos′, satisfy the axioms $\mathcal{K}$, where $\mathcal{K} \in \{\mathcal{K}_f, \mathcal{K}_v\}$, then pos′ satisfies the strict monotonicity condition Mon(pos′).

Checking (a) is not a problem. For (b) we need to show that in the extension $\mathcal{T}$ of a combination $\mathcal{T}_0$ of real arithmetic with an index theory describing precedence of trains, with the two functions pos and pos' the following hold:

$$\mathcal{T} \models \mathcal{K} \wedge \mathsf{Mon}(\mathsf{pos}) \rightarrow \mathsf{Mon}(\mathsf{pos}'), \quad \text{i.e.} \quad \mathcal{T} \wedge \mathcal{K} \wedge \mathsf{Mon}(\mathsf{pos}) \wedge \neg \mathsf{Mon}(\mathsf{pos}') \models \bot .$$

The set of formulae to be proved unsatisfiable w.r.t. $\mathcal{T}$ involves the axioms $\mathcal{K}$ and $\mathsf{Mon}(\mathsf{pos})$, containing universally quantified variables of sort i. Only $\neg\mathsf{Mon}(\mathsf{pos}')$ corresponds to a ground set of clauses G. However, positive results for reasoning in combinations of theories were only obtained for testing satisfiability for ground formulae [7, 4], so are not directly applicable.

In bounded model checking the same problem occurs. For a fixed k, one has to show that there are no paths of length at most k from the initial state to an unsafe state. We therefore need to store all intermediate positions in arrays $\mathsf{pos}_0, \mathsf{pos}_1, \ldots, \mathsf{pos}_k$, and – provided that $\mathcal{K}(\mathsf{pos}_{i-1}, \mathsf{pos}_i)$ is defined such that $\mathcal{K} = \mathcal{K}(\mathsf{pos}, \mathsf{pos}')$ – to show:

$$\mathcal{T} \wedge \bigwedge_{i=1}^{j} \mathcal{K}(\mathsf{pos}_{i-1}, \mathsf{pos}_i) \wedge \mathsf{Mon}(\mathsf{pos}_0) \wedge \neg \mathsf{Mon}(\mathsf{pos}_j) \models \bot \qquad \text{for all } 0 \leq j \leq k.$$

4.2 Our solution: locality, hierarchical reasoning

Our idea. In order to overcome the problem mentioned above we proceed as follows. We consider two successive extensions of the base many-sorted combination $\mathcal{T}_0$ of real arithmetic (for reasoning about positions, sort num) with an index theory (for describing precedence between trains, sort i):

- the extension $\mathcal{T}_1$ of $\mathcal{T}_0$ with a monotone function pos, of arity $\mathsf{i} \rightarrow \mathsf{num}$,
- the extension $\mathcal{T}_2$ of $\mathcal{T}_1$ with a function pos' satisfying $\mathcal{K} \in \{\mathcal{K}_f, \mathcal{K}_v\}$.

We show that both extensions $\mathcal{T}_0 \subseteq \mathcal{T}_1 = \mathcal{T}_0 \cup \mathsf{Mon}(\mathsf{pos})$ and $\mathcal{T}_1 \subseteq \mathcal{T}_2 = \mathcal{T}_1 \cup \mathcal{K}$ are local, where $\mathcal{K} \in \{\mathcal{K}_f, \mathcal{K}_v\}$. This allows us to reduce problem (b) to testing satisfiability of ground clauses in $\mathcal{T}_0$, for which standard methods for reasoning in combinations of theories can be applied. A similar method can be used for bounded model checking.

The base theory. As mentioned before, we assume that $\mathcal{T}_0$ is the many-sorted combination of a theory $\mathcal{T}_0^{\mathsf{i}}$ (sort i) for reasoning about precedence between trains and the theory $\mathcal{T}_0^{\mathsf{num}}$ of real numbers (sort num) for reasoning about distances between trains. As a convention, everywhere in what follows i, j, k denote variables of sort i and c, d denote variables of sort num.

We have several possibilities of choosing $\mathcal{T}_0^{\mathsf{i}}$: we can model the trains on a track by using an (acyclic) list structure, where any train is linked to its predecessor, or using the theory of integers with predecessor.

In what follows let $\mathcal{T}_0^{\mathsf{i}}$ be (a fragment of) integer arithmetic and $\mathcal{T}_0^{\mathsf{num}}$ be the theory of real or rational numbers. In both these theories satisfiability of ground clauses is decidable.

Collision freeness as monotonicity. Let $\mathcal{T}_0$ be the (disjoint, many-sorted) combination of $\mathcal{T}_0^{\mathsf{i}}$ and $\mathcal{T}_0^{\mathsf{num}}$. Then classical methods on combinations of decision procedures for (disjoint, many-sorted) theories can be used to give a decision procedure for satisfiability of ground clauses w.r.t. $\mathcal{T}_0$. Let $\mathcal{T}_1$ be obtained by extending $\mathcal{T}_0$ with a function pos of arity $\mathsf{i} \to \mathsf{num}$ mapping train indices to the real numbers, which satisfies condition $\mathsf{Mon}(\mathsf{pos})$:

$$\mathsf{Mon}(\mathsf{pos}) \quad \forall i, j \;\; (\mathsf{first} \leq i < j \leq \mathsf{last} \rightarrow \mathsf{pos}(i) >_{\mathbb{R}} \mathsf{pos}(j)),$$

where i and j are indices, $<$ is the ordering on indices and $>_{\mathbb{R}}$ is the usual ordering on the real numbers. (For the case of a fixed number of trains, we can assume that $\mathsf{first} = 0$ and $\mathsf{last} = n - 1$.)

A more precise axiomatization of collision-freeness. The monotonicity axiom above is, in fact, an oversimplification. A more precise model, in which the length of trains is considered can be obtained by replacing the monotonicity axiom for pos with the following axiom:

$$\forall i, j, k \quad (\mathsf{first} \leq j \leq i \leq \mathsf{last} \wedge i - j = k \rightarrow \mathsf{pos}(j) - \mathsf{pos}(i) \geq k * \mathsf{LengthTrain}),$$

where $\mathsf{LengthTrain}$ is the standard (resp. maximal) length of a train.

As base theory we consider the combination $\mathcal{T}_0'$ of the theory of integers and reals with a multiplication operation $*$ of arity $\mathsf{i} \times \mathsf{num} \to \mathsf{num}$ (multiplication of k with the constant $\mathsf{LengthTrain}$ in the formula above)[3].

Let $\mathcal{T}_1'$ be the theory obtained by extending the combination $\mathcal{T}_0'$ of the theory of integers and reals with a function pos satisfying the axiom above.

Theorem 2 *The following extensions are local theory extensions:*

(1) The theory extension $\mathcal{T}_0 \subseteq \mathcal{T}_1$.
(2) The theory extension $\mathcal{T}_0' \subseteq \mathcal{T}_1'$.

Proof: We prove that every model of $\mathcal{T}_1$ in which the function pos is partially defined can be extended to a model in which pos is totally defined. Locality then follows by results in [8]. To define pos at positions where it is undefined we use the density of real numbers and the discreteness of the index theory (between two integers there are only finitely many integers). □

We now extend the resulting theory $\mathcal{T}_1$ again in two different ways, with the axiom sets for one of the two system models, respectively. A similar construction can be done starting from the theory $\mathcal{T}_1'$.

Theorem 3 *The following extensions are local theory extensions:*

(1) The extension $\mathcal{T}_1 \subseteq \mathcal{T}_1 \cup \mathcal{K}_f$
(2) The extension $\mathcal{T}_1 \subseteq \mathcal{T}_1 \cup \mathcal{K}_v$.

[3]In the light of locality properties of such extensions (cf. Theorem 2), k will always be instantiated by values in a finite set of *concrete* integere, all within a given, *concrete* range; thus the introduction of this many-sorted multiplication does not affect decidability.

Proof: (1) Clauses in $\mathcal{K}_f$ are flat and linear w.r.t. pos', so we again prove locality of the extension by showing that weak partial models can be extended to total ones. The proof proceeds by a case distinction. We use the fact that the left-hand sides of the implications in $\mathcal{K}_f$ are mutually exclusive. (2) is proved similarly. □

Let $\mathcal{K} \in \{\mathcal{K}_v, \mathcal{K}_f\}$. By the locality of $\mathcal{T}_1 \subseteq \mathcal{T}_2 = \mathcal{T}_1 \cup \mathcal{K}$ and by Theorem 1, the following are equivalent:

(1) $\mathcal{T}_0 \wedge \mathsf{Mon}(\mathsf{pos}) \wedge \mathcal{K} \wedge \neg\mathsf{Mon}(\mathsf{pos}') \models \bot$,
(2) $\mathcal{T}_0 \wedge \mathsf{Mon}(\mathsf{pos}) \wedge \mathcal{K}[G] \wedge G \models_w \bot$, where $G = \neg\mathsf{Mon}(\mathsf{pos}')$,
(3) $\mathcal{T}_0 \wedge \mathsf{Mon}(\mathsf{pos}) \wedge \mathcal{K}_0 \wedge G_0 \wedge N_0(\mathsf{pos}') \models \bot$,

where $\mathcal{K}[G]$ consists of all instances of the rules in $\mathcal{K}$ in which the terms starting with the function symbols pos' are ground subterms already occurring in G or $\mathcal{K}$, $\mathcal{K}_0 \wedge G_0$ is obtained from $\mathcal{K}[G] \wedge G$ by introducing new constants for the arguments of the extension functions as well as for the (sub)terms $t = f(g_1, \ldots, g_n)$ starting with extension functions $f \in \Sigma_1$, and $N_0(\mathsf{pos}')$ is the set of instances of the congruence axioms for pos' which correspond to the definitions for these newly introduced constants.

It is easy to see that, due to the special form of the rules in $\mathcal{K}$ (all free variables in any clause occur as arguments of pos' both in $\mathcal{K}_f$ and in $\mathcal{K}_v$), $\mathcal{K}[G]$ (hence also $\mathcal{K}_0$) is a set of ground clauses. By the locality of $\mathcal{T}_0 \subseteq \mathcal{T}_1 = \mathcal{T}_0 \cup \mathsf{Mon}(\mathsf{pos})$, the following are equivalent:

(1) $\mathcal{T}_0 \wedge \mathsf{Mon}(\mathsf{pos}) \wedge \mathcal{K}_0 \wedge G_0 \wedge N_0(\mathsf{pos}') \models \bot$,
(2) $\mathcal{T}_0 \wedge \mathsf{Mon}(\mathsf{pos})[G'] \wedge G' \models_w \bot$, where $G' = \mathcal{K}_0 \wedge G_0 \wedge N_0(\mathsf{pos}')$,
(3) $\mathcal{T}_0 \wedge \mathsf{Mon}(\mathsf{pos})_0 \wedge G'_0 \wedge N_0(\mathsf{pos}) \models \bot$,

where $\mathsf{Mon}(\mathsf{pos})[G']$ consists of all instances of the rules in $\mathsf{Mon}(\mathsf{pos})$ in which the terms starting with the function symbol pos are ground subterms already occurring in G', $\mathsf{Mon}(\mathsf{pos})_0 \wedge G'_0$ is obtained from $\mathsf{Mon}(\mathsf{pos})[G'] \wedge G'$ by purification and flattening, and $N_0(\mathsf{pos})$ corresponds to the set of instances of congruence axioms for pos which need to be taken into account.

This allows us to use hierarchical reasoning on properties of the system, i.e. to reduce the verification of system properties to deciding satisfiability of constraints in $\mathcal{T}_0$. An advantage is that, after the reduction of the problem to a satisfiability problem in the base theory, one can automatically determine which constraints on the parameters (e.g. $\Delta\mathsf{t}, \mathsf{min}, \mathsf{max}, \ldots$) guarantee truth of the invariant. This can be achieved, e.g. using quantifier elimination. The method is illustrated in Section 4.3; more details can be found in [5].

Similar results can be established for bounded model checking. In this case the arguments are similar, but one needs to consider chains of extensions of length $1, 2, 3, \ldots, k$ for a bounded k, corresponding to the paths from

the initial state to be analyzed. An interesting side-effect of our approach (restricting to instances which are similar to the goal) is that it provides a possibility of systematic "slicing": for proving (disproving) the violation of the safety condition we only need to consider those trains which are in a 'neighborhood' of the trains which violate the safety condition.

4.3 Illustration

In this section we indicate how to apply hierarchical reasoning on the case study given in Section 3, Model 1. We follow the steps given at the end of Section 2 and show how the sets of formulas are obtained that can finally be handed to a prover of the base theory $\mathcal{T}_0$.

To check whether $\mathcal{T}_1 \cup \mathcal{K}_f \models \mathsf{ColFree}(\mathsf{pos}')$, where

$$\mathsf{ColFree}(\mathsf{pos}') \quad \forall i \quad (0 \leq i < \mathsf{n} - 1 \quad \rightarrow \quad \mathsf{pos}'(i) >_{\mathbb{R}} \mathsf{pos}'(i+1)),$$

we check whether $\mathcal{T}_1 \cup \mathcal{K}_f \cup G \models \bot$, where $G = \{0 \leq k < \mathsf{n} - 1, \quad k' = k + 1, \quad \mathsf{pos}'(k) \leq_{\mathbb{R}} \mathsf{pos}'(k')\}$ is the (skolemized) negation of $\mathsf{ColFree}(\mathsf{pos}')$, flattened by introducing a new constant k'. This problem is reduced to a satisfiability problem over $\mathcal{T}_1$ as follows:

Step 1: Use locality. We construct the set $\mathcal{K}_f[G]$: There are no ground subterms with pos' at the root in $\mathcal{K}_f$, and only two ground terms with pos' in G, $\mathsf{pos}'(k)$ and $\mathsf{pos}'(k')$. This means that $\mathcal{K}_f[G]$ consists of two instances of $\mathcal{K}_f$: one with i instantiated to k, the other instantiated to k'. E.g., the two instances of $\mathsf{F2}$ are:

$$\begin{aligned}(\mathsf{F2[G]}) \quad & (0 < k < \mathsf{n} \quad \wedge \quad \mathsf{pos}(k-1) >_{\mathbb{R}} 0 \quad \wedge \quad \mathsf{pos}(k-1) - \mathsf{pos}(k) \geq_{\mathbb{R}} \mathsf{l_{alarm}} \\ & \quad \rightarrow \mathsf{pos}(k) + \Delta\mathsf{t} * \mathsf{min} \leq_{\mathbb{R}} \mathsf{pos}'(k) \leq_{\mathbb{R}} \mathsf{pos}(k) + \Delta\mathsf{t}{*}\mathsf{max}) \\ & (0 < k' < \mathsf{n} \quad \wedge \quad \mathsf{pos}(k'-1) >_{\mathbb{R}} 0 \quad \wedge \quad \mathsf{pos}(k'-1) - \mathsf{pos}(k') \geq_{\mathbb{R}} \mathsf{l_{alarm}} \\ & \quad \rightarrow \mathsf{pos}(k') + \Delta\mathsf{t} * \mathsf{min} \leq_{\mathbb{R}} \mathsf{pos}'(k') \leq_{\mathbb{R}} \mathsf{pos}(k') + \Delta\mathsf{t}{*}\mathsf{max})\end{aligned}$$

The construction of $(\mathsf{F1[G]})$, $(\mathsf{F3[G]})$ and $(\mathsf{F4[G]})$ is similar. In addition, we specify the known relationships between the constants of the system:
(Dom) $\Delta\mathsf{t} > 0 \quad \wedge \quad 0 \leq \mathsf{min} \quad \wedge \quad \mathsf{min} \leq \mathsf{max}$

Step 2: Flattening and purification. $\mathcal{K}_f[G] \wedge G$ is already flat w.r.t. pos'. We replace all ground terms with pos' at the root with new constants: we replace $\mathsf{pos}'(k)$ by c_1 and $\mathsf{pos}'(k')$ by c_2. We obtain a set of definitions $D = \{\mathsf{pos}'(k) = c_1, \mathsf{pos}'(k') = c_2\}$ and a set $\mathcal{K}_{f_0}$ of clauses which do not contain occurrences of pos', consisting of (Dom) together with:

$$(\mathsf{G_0}) \qquad 0 \leq k < \mathsf{n} - 1 \quad \wedge \quad k' = k + 1 \quad \wedge \quad c_1 \leq_{\mathbb{R}} c_2$$

$$\begin{aligned}(\mathsf{F2_0}) \quad & (0 < k < \mathsf{n} \quad \wedge \quad \mathsf{pos}(k-1) >_{\mathbb{R}} 0 \quad \wedge \quad \mathsf{pos}(k-1) - \mathsf{pos}(k) \geq_{\mathbb{R}} \mathsf{l_{alarm}} \\ & \quad \rightarrow \mathsf{pos}(k) + \Delta\mathsf{t} * \mathsf{min} \leq_{\mathbb{R}} c_1 \leq_{\mathbb{R}} \mathsf{pos}(k) + \Delta\mathsf{t}{*}\mathsf{max}) \\ & (0 < k' < \mathsf{n} \quad \wedge \quad \mathsf{pos}(k'-1) >_{\mathbb{R}} 0 \quad \wedge \quad \mathsf{pos}(k'-1) - \mathsf{pos}(k') \geq_{\mathbb{R}} \mathsf{l_{alarm}} \\ & \quad \rightarrow \mathsf{pos}(k') + \Delta\mathsf{t} * \mathsf{min} \leq_{\mathbb{R}} c_2 \leq_{\mathbb{R}} \mathsf{pos}(k') + \Delta\mathsf{t}{*}\mathsf{max})\end{aligned}$$

The construction can be continued similarly for $\mathsf{F1}$, $\mathsf{F3}$ and $\mathsf{F4}$.

Step 3: Reduction to satisfiability in $\mathcal{T}_1$. We add the functionality clause $N_0 = \{k = k' \rightarrow c_1 = c_2\}$ and obtain a satisfiability problem in $\mathcal{T}_1$: $\mathcal{K}_{f_0} \wedge G_0 \wedge N_0$. To decide satisfiability of $\mathcal{T}_1 \wedge \mathcal{K}_{f_0} \wedge G_0 \wedge N_0$, we have to do another transformation w.r.t. the extension $\mathcal{T}_0 \subseteq \mathcal{T}_1$. The resulting set of ground clauses can directly be handed to a decision procedure for the combination of the theory of indices and the theory of reals. We flatten and purify the set $\mathcal{K}_{f_0} \wedge G_0 \wedge N_0$ of ground clauses w.r.t. pos by introducing new constants denoting $k-1$ and $k'-1$, together with their definitions $k'' = k-1, k''' = k'-1$; as well as constants d_i for $\mathsf{pos}(k), \mathsf{pos}(k'), \mathsf{pos}(k''), \mathsf{pos}(k''')$. Taking into account only the corresponding instances of the monotonicity axiom for pos we obtain a set of clauses consisting of (Dom) together with:

$$
\begin{array}{ll}
(\mathsf{G}_0') & k'' = k-1 \ \wedge \ k''' = k'-1 \\
(\mathsf{G}_0) & 0 \leq k < n-1 \ \wedge \ k' = k+1 \ \wedge \ c_1 \leq_{\mathbb{R}} c_2 \\
(\mathsf{GF2}_0) & 0{<}k{<}n \ \wedge \ d_3{>}_{\mathbb{R}}0 \ \wedge \ d_3{-}d_1 \geq_{\mathbb{R}} \mathsf{l_{alarm}} \rightarrow d_1{+}\Delta\mathsf{t}{*}\mathsf{min} \leq_{\mathbb{R}} c_1 \leq_{\mathbb{R}} d_1{+}\Delta\mathsf{t}{*}\mathsf{max} \\
 & 0{<}k'{<}n \ \wedge \ d_4{>}_{\mathbb{R}}0 \ \wedge \ d_4{-}d_2 \geq_{\mathbb{R}} \mathsf{l_{alarm}} \rightarrow d_2{+}\Delta\mathsf{t}{*}\mathsf{min} \leq_{\mathbb{R}} c_2 \leq_{\mathbb{R}} d_2{+}\Delta\mathsf{t}{*}\mathsf{max}
\end{array}
$$

and $\mathsf{Mon}(\mathsf{pos})[G']$. After making some simplifications we obtain the following set of (many-sorted) constraints:

$C_{\mathsf{Definitions}}$		C_{Indices}	C_{Reals}		C_{Mixed}
$\mathsf{pos}'(k) = c_1$	$\mathsf{pos}(k') = d_2$	$k'' = k-1$	$c_1 \leq_{\mathbb{R}} c_2$	$d_3 >_{\mathbb{R}} d_4$	$(\mathsf{GF1}_0)$
$\mathsf{pos}'(k') = c_2$	$\mathsf{pos}(k'') = d_3$	$k''' = k'-1$	$d_1 >_{\mathbb{R}} d_2$	$d_4 >_{\mathbb{R}} d_2$	$(\mathsf{GF2}_0)$
$\mathsf{pos}(k) = d_1$	$\mathsf{pos}(k''') = d_4$	$0 \leq k < n-1$	$d_3 >_{\mathbb{R}} d_1$	$d_1 = d_4$	$(\mathsf{GF3}_0)$
		$k' = k+1$	$d_3 >_{\mathbb{R}} d_2$	(Dom)	$(\mathsf{GF4}_0)$

For checking the satisfiability of $C_{\mathsf{Indices}} \wedge C_{\mathsf{Reals}} \wedge C_{\mathsf{Mixed}}$ we can use a prover for the two-sorted combination of the theory of integers and the theory of reals, possibly combined with a DPLL methodology for dealing with full clauses. An alternative method, somewhat similar to DPLL($\mathcal{T}_0$), would be to use only branching on the literals containing terms of index sort – this reduces the verification problem to the problem of checking the satisfiability of a set of linear constraints over the reals.

5 Conclusions

In this paper we described a case study concerning a system of trains on a rail track, where trains may enter and leave the area. An example of a safety condition for such a system (collision freeness) was considered. The problem above can be reduced to testing satisfiability of *quantified formulae* in complex theories. However, the existing results on reasoning in combinations of theories are restricted to testing satisfiability for *ground formulae*.

This paper shows that, in the example considered, we can reduce satisfiability checking of universally quantified formulae to the simpler task of satisfiability checking for ground clauses. For this, we identify corresponding chains of theory extensions $\mathcal{T}_0 \subseteq \mathcal{T}_1 \subseteq \cdots \subseteq \mathcal{T}_i$, such that $\mathcal{T}_j = \mathcal{T}_{j-1} \cup \mathcal{K}_j$ is a local extension of $\mathcal{T}_{j-1}$ by a set $\mathcal{K}_j$ of (universally quantified) clauses. This

allows us to reduce, for instance, testing collision freeness in theories containing arrays to represent the train positions, to checking the satisfiability of a set of sets of ground clauses over the combination of the theory of reals with a theory which expresses precedence between trains. The applicability of the method is however general: the challenge is, at the moment, to recognize classes of local theories occurring in various areas of application. The implementation of the procedure described here is in progress, the method is clearly easy to implement. At a different level, our results open a possibility of using abstraction-refinement deductive model checking in a whole class of applications including the examples presented here – these aspects are not discussed in this paper, and rely on results we obtained in [9].

The results we present here also have theoretical implications: In one of the models we considered here, collision-freeness is expressed as a monotonicity condition. Limits of decidability in reasoning about sorted arrays were explored in [1]. The decidability of satisfiability of ground clauses in the fragment of the theory of sorted arrays which we consider here is an easy consequence of the locality of extensions with monotone functions.

Acknowledgements. This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS). See `www.avacs.org` for more information.

References

[1] A. Bradley, Z. Manna, and H. Sipma. What's decidable about arrays? In E. Emerson and K. Namjoshi, editors, *Verification, Model-Checking, and Abstract-Interpretation, 7th Int. Conf. (VMCAI 2006)*, LNCS 3855, pp. 427–442. Springer, 2006.

[2] J. Faber. Verifying real-time aspects of the European Train Control System. In *Proceedings of the 17th Nordic Workshop on Programming Theory*, pp. 67–70. University of Copenhagen, Denmark, October 2005.

[3] H. Ganzinger, V. Sofronie-Stokkermans, and U. Waldmann. Modular proof systems for partial functions with weak equality. In D. Basin and M. Rusinowitch, editors, *Automated reasoning : 2nd Int. Joint Conference, IJCAR 2004*, LNAI 3097, pp. 168–182. Springer, 2004. An extended version will appear in Information and Computation.

[4] S. Ghilardi. Model theoretic methods in combined constraint satisfiability. *Journal of Automated Reasoning*, 33(3–4):221–249, 2004.

[5] S. Jacobs and V. Sofronie-Stokkermans. Applications of hierarchical reasoning in the verification of complex systems (extended version). Available online at `http://www.mpi-sb.mpg.de/~sofronie/papers/jacobs-sofronie-pdpar-extended.ps`

[6] S. McPeak and G. Necula. Data structure specifications via local equality axioms. In K. Etessami and S. Rajamani, editors, *Computer Aided Verification, 17th International Conference, CAV 2005*, LNCS 3576, pp. 476–490, 2005.

[7] G. Nelson and D. Oppen. Simplification by cooperating decision procedures. *ACM Trans. on Programming Languages and Systems*, 1(2):245–257, 1979.

[8] V. Sofronie-Stokkermans. Hierarchic reasoning in local theory extensions. *CADE'2005: Int. Conf. on Automated Deduction*, LNCS 3632, pp. 219–234. Springer, 2005.

[9] V. Sofronie-Stokkermans. Interpolation in local theory extensions. In *Proceedings of IJCAR 2006*, LNAI 4130, pp. 235–250. Springer, 2006.

Towards Automatic Proofs of Inequalities Involving Elementary Functions

Behzad Akbarpour and Lawrence C. Paulson

Computer Laboratory, University of Cambridge

Abstract

Inequalities involving functions such as sines, exponentials and logarithms lie outside the scope of decision procedures, and can only be solved using heuristic methods. Preliminary investigations suggest that many such problems can be solved by reduction to algebraic inequalities, which can then be decided by a decision procedure for the theory of real closed fields (RCF). The reduction involves replacing each occurrence of a function by a lower or upper bound (as appropriate) typically derived from a power series expansion. Typically this requires splitting the domain of the function being replaced, since most bounds are only valid for specific intervals.

1 Introduction

Decision procedures are valuable, but too many problems lie outside of their scope. Linear arithmetic restricts us to the language of $=$, $<$, $\leq$, $+$ and multiplication by integer constants, combined by Boolean connectives. In their formalization of the Prime Number Theorem [2], Avigad and his colleagues spent much time proving simple facts involving logarithms. We would like to be able to prove inequalities involving any of the so-called elementary functions: sine, cosine, arctangent, logarithm and exponential. Richardson's theorem tells us that this problem is undecidable [11], so we are left with heuristic methods.

In this paper, we outline preliminary work towards such heuristics. We have no implementation nor even a definite procedure, but we do have methods that we have tested by hand on about 30 problems. Our starting point is that the theory of real closed fields—that is, the real numbers with addition and multiplication—is decidable. Our idea is to replace each occurrence of an elementary function by an algebraic expression that is known to be an upper or lower bound, as appropriate. If this results in a purely algebraic inequality, then we supply the problem to a decision procedure for the theory of real closed fields.

Complications include the need for case analysis on the arguments of elementary functions, since many bounds are only valid over restricted intervals. If these arguments are complex expressions, then identifying their range requires something like a recursive application of the method. The resulting algebraic inequalities may be too difficult to be solved efficiently. Even so, the method works on many problems.

Paper outline. We begin by reviewing the basis of our work, namely existing decision procedures for polynomials and prior work on verifying inequalities involving the elementary functions (Sect. 2). To illustrate the idea, we present a simple example involving the exponential function (Sect. 3) and then a more complex example involving the logarithm function (Sect. 4). We conclude by presenting a list of solved problems and outlining our next steps (Sect. 5).

2 Background

Our work relies on the existence of practical, if not always efficient, decision procedures for the theory of real closed fields (RCF). According to Dolzmann et al. [3], Tarski found the first quantifier elimination procedure in the 1930s, while Collins introduced the first feasible method in 1975. His *cylindrical algebraic decomposition* is still doubly exponential in the worst case. Dolzmann et al. proceed to survey several quantifier elimination algorithms and their applications. One freely-available implementation is QEPCAD [5], a decision procedure that performs partial cylindrical algebraic decomposition. The prover HOL Light provides a simpler quantifier elimination procedure for real closed fields [8]. Also in HOL Light is an implementation of Parrilo's method [10] for deciding polynomials using sum-of-squares decompositions; less general than any quantifier elimination procedure, it is dramatically more efficient.[1] Some polynomial inequalities can also be tackled using heuristic procedures such as those of Hunt et al. [6] and Tiwari [12].

Our second starting point is the work of Mũnoz and Lester [9], on proving real number inequalities that may contain the elementary functions, but no variables. The example they give is

$$\frac{3\pi}{180} \leq \frac{g}{v} \tan\left(\frac{35\pi}{180}\right),$$

where g and v are constants. Their method for proving such ground inequalities relies on upper and lower bounds for the elementary functions, coupled with interval arithmetic. The absence of variables makes the problem much simpler; in particular, if we need to establish the range of the argument x in $\tan(x)$, we simply call the procedure recursively.

[1] Harrison has mentioned this implementation [4], but as yet no documentation exists.

These methods might be expected to work for some problems containing variables. Interval arithmetic should be able to prove some inequalities involving a variable x say, if we know that $0 \leq x \leq 1$. However, the method fails on some easy-looking problems; as Mũnoz and Lester note, interval arithmetic can lose information rapidly. For example, if $x \in [0,1]$, interval arithmetic cannot prove the trivial $x - x \geq 0$: we get $[0,1] - [0,1] = [0,1] + [-1,0] = [-1,1]$, and neither $[-1,1] \leq 0$ nor $[-1,1] \geq 0$ hold. This is a well-known issue and there are some techniques that can reduce its impact, such as (obviously) reducing $x - x$ to 0 before applying interval arithmetic. But, in some cases, when we wanted to prove $E \leq 0$, the best we could do with interval arithmetic was to prove that $E \leq \epsilon$ for an arbitrary small, but positive, ϵ. A method based on a decision procedure for real closed fields ought to be more general and effective.

3 A Simple Example Concerning Exponentials

Figure 1 presents a family of upper and lower bounds for the exponential function. Mũnoz and Lester [9] give similar bounds, but we have corrected errors in the first two and altered the presentation. All conventions are as in the original paper. The lower bound is written $\underline{\exp}(x,n)$ and the upper bound is written $\overline{\exp}(x,n)$, where n is a non-negative integer. For all x and n, they satisfy

$$\underline{\exp}(x,n) \leq e^x \leq \overline{\exp}(x,n).$$

As n increases, the bounds converge monotonically to the target function, here exp. As n increases, the bounds get tighter and the RCF problems that must be decided get harder; in return, we should be able to prove harder inequalities involving exponentials.

Case analysis on the value of x in e^x cannot be avoided. Clearly no polynomial could serve as an upper bound, or as an accurate lower bound, of the exponential function. The role of m in these bounds is to segment the real line into integers, with separate bounds in each segment. These case analyses will complicate our proofs. In particular, unless the argument of the exponential function has a finite range, these bounds are useless, since they would require the examination of infinitely many cases.

For a simple demonstration of our idea, let us prove the theorem

$$0 \leq x \leq 1 \Longrightarrow e^x \leq 1 + x + x^2.$$

Here it suffices to replace the function e by an upper bound:

$$0 \leq x \leq 1 \Longrightarrow \overline{\exp}(x,n) \leq 1 + x + x^2.$$

We have a lower bound if $0 < x \leq 1$, so we need to perform a simple case analysis.

$$\underline{\exp}(x,n) = \sum_{i=0}^{2(n+1)+1} \frac{x^i}{i!} \qquad \text{if } -1 \le x < 0 \tag{1}$$

$$\overline{\exp}(x,n) = \sum_{i=0}^{2(n+1)} \frac{x^i}{i!} \qquad \text{if } -1 \le x < 0 \tag{2}$$

$$\underline{\exp}(0,n) = \overline{\exp}(0,n) = 1 \tag{3}$$

$$\underline{\exp}(x,n) = \frac{1}{\overline{\exp}(-x,n)} \qquad \text{if } 0 < x \le 1 \tag{4}$$

$$\overline{\exp}(x,n) = \frac{1}{\underline{\exp}(-x,n)} \qquad \text{if } 0 < x \le 1 \tag{5}$$

$$\underline{\exp}(x,n) = \underline{\exp}(x/m,n)^m \quad \text{if } \ x < -1,\ m = -\lfloor x \rfloor \tag{6}$$

$$\overline{\exp}(x,n) = \overline{\exp}(x/m,n)^m \quad \text{if } \ x < -1,\ m = -\lfloor x \rfloor \tag{7}$$

$$\underline{\exp}(x,n) = \overline{\exp}(x/m,n)^m \quad \text{if } \ 1 < x,\ m = \lfloor -x \rfloor \tag{8}$$

$$\overline{\exp}(x,n) = \underline{\exp}(x/m,n)^m \quad \text{if } \ 1 < x,\ m = \lfloor -x \rfloor \tag{9}$$

Figure 1: Bounds for the Exponential Function

- If $x = 0$ then $\overline{\exp}(0, n) = 1 \leq 1 + 0 + 0^2 = 1$, trivially.
- If $0 < x \leq 1$, then by equations (5) and (1)

$$\overline{\exp}(x, n) = \left(\sum_{i=0}^{2(n+1)+1} \frac{(-x)^i}{i!} \right)^{-1}$$

and putting $n = 0$, it suffices to prove

$$\left(1 + (-x) + \frac{(-x)^2}{2} + \frac{(-x)^3}{6}\right)^{-1} \leq 1 + x + x^2.$$

This last inequality is non-trivial, but as it falls within RCF, it can be proved automatically. Existing tools require us first to eliminate the division, reducing the problem to the two inequalities

$$0 < 1 - x + \frac{x^2}{2} - \frac{x^3}{6} \quad \text{and} \quad 1 \leq \left(1 + x + x^2\right)\left(1 - x + \frac{x^2}{2} - \frac{x^3}{6}\right).$$

HOL Light has two separate tools that can prove these. Sean McLaughlin's quantifier elimination package [8] can prove the pair of inequalities in 351 seconds, while John Harrison's implementation of the sum-of-squares method [10] needs only 0.48 seconds.[2]

Let us check these inequalities ourselves. The first one is clear, since $x^{k+1} \leq x^k$ for all k. Multiplying out the second inequality reduces it to

$$0 \leq \frac{x^2}{2} - \frac{2x^3}{3} + \frac{x^4}{3} - \frac{x^5}{6}.$$

Multiplying both sides by 6 and factoring reduces this inequality to

$$0 \leq x^2(1 - x)(3 - x + x^2)$$

when it is obvious that all of the factors are non-negative.

This proof is not obvious, and its length shows that we have much to gain by automating the procedure. That involves performing the case analysis, substituting the appropriate bounds, calling an RCF decision procedure, and in case of failure, retrying with a larger value of n.

4 An Extended Example Concerning Logarithms

Figure 2 presents the bounds for the logarithm function. They are again taken from Mũnoz and Lester [9], while correcting several errata. The next

[2] All timings were done on a dual 3GHz Pentium equipped with 4GB of memory.

example will demonstrate how a complicated derivation can arise from a simple-looking inequality:

$$-\frac{1}{2} < x \leq 3 \Longrightarrow \ln(1+x) \leq x.$$

We re-express the condition on x in terms of $1+x$, which is the argument of ln, when substituting in the lower bound:

$$\frac{1}{2} < 1+x \leq 4 \Longrightarrow \overline{\ln}(1+x,n) \leq x$$

As with the exponential function, to obtain reasonably tight bounds requires considering rather small intervals. Our problem splits into four cases:

$$\frac{1}{2} < 1+x < 1 \quad \text{or} \quad 1+x = 1 \quad \text{or} \quad 1 < 1+x \leq 2 \quad \text{or} \quad 2 < 1+x \leq 4$$

Let us leave the first case for last, as it is the most complicated, and consider the other three cases.

$$\underline{\ln}(x,n) = \sum_{i=1}^{2n} (-1)^{i+1} \frac{(x-1)^i}{i} \qquad \text{if } 1 < x \leq 2 \tag{10}$$

$$\overline{\ln}(x,n) = \sum_{i=1}^{2n+1} (-1)^{i+1} \frac{(x-1)^i}{i} \qquad \text{if } 1 < x \leq 2 \tag{11}$$

$$\underline{\ln}(1,n) = \overline{\ln}(1,n) = 0 \tag{12}$$

$$\underline{\ln}(x,n) = -\overline{\ln}\left(\frac{1}{x}, n\right), \qquad \text{if } 0 < x < 1 \tag{13}$$

$$\overline{\ln}(x,n) = -\underline{\ln}\left(\frac{1}{x}, n\right), \qquad \text{if } 0 < x < 1 \tag{14}$$

$$\underline{\ln}(x,n) = m\ \underline{\ln}(2,n)\ +\ \underline{\ln}(y,n) \quad \text{if } x > 2,\ x = 2^m y,\ 1 < y \leq 2 \tag{15}$$

$$\overline{\ln}(x,n) = m\ \overline{\ln}(2,n)\ +\ \overline{\ln}(y,n) \quad \text{if } x > 2,\ x = 2^m y,\ 1 < y \leq 2 \tag{16}$$

Figure 2: Bounds for the Logarithm Function

If $1+x = 1$, then $x = 0$ and trivially $\overline{\ln}(1+x,n) = \overline{\ln}(1,n) = 0 \leq x$.
If $1 < 1+x \leq 2$, then

$$\overline{\ln}(1+x,n) = \sum_{i=1}^{2n+1} (-1)^{i+1} \frac{((1+x)-1)^i}{i} = \sum_{i=1}^{2n+1} (-1)^{i+1} \frac{x^i}{i}$$

by equation (11). Setting $n = 0$ yields $\overline{\ln}(1 + x, n) = x$ and reduces our inequality to the trivial $x \leq x$.

If $2 < 1 + x \leq 4$, then we have to apply equation (16). That requires finding a positive integer m and some y such that $1+x = 2^m y$ and $1 < y \leq 2$. Clearly $m = 1$. In this case, putting $n = 0$, we have

$$\begin{aligned}\sum_{i=1}^{2n+1} (-1)^{i+1} \frac{(2-1)^i}{i} + \sum_{i=1}^{2n+1} (-1)^{i+1} \frac{(y-1)^i}{i} &= 1 + (y-1) \\ &= y \\ &\leq 2y - 1 \\ &= x.\end{aligned}$$

Now, let us turn to that postponed first case. If $\frac{1}{2} < 1 + x < 1$, then $1 < 1/(1 + x) < 2$. Putting $n = 1$, we have

$$\begin{aligned}\overline{\ln}(1 + x, n) &= -\underline{\ln}\left(\frac{1}{1+x}, n\right) \\ &= -\sum_{i=1}^{2n} (-1)^{i+1} \frac{\left(\frac{1}{1+x} - 1\right)^i}{i} \\ &= \sum_{i=1}^{2n} \frac{(-1)^i}{i} \left(\frac{-x}{1+x}\right)^i \\ &= \left(\frac{x}{1+x}\right) + \left(\frac{1}{2}\right)\left(\frac{-x}{1+x}\right)^2.\end{aligned}$$

Now

$$\begin{aligned}\left(\frac{x}{1+x}\right) + \left(\frac{1}{2}\right)\left(\frac{-x}{1+x}\right)^2 \leq x &\iff \\ x(1+x) + \frac{1}{2}x^2 \leq x(1+x)^2 &\iff \\ x + \frac{3}{2}x^2 \leq x + 2x^2 + x^3 &\iff \\ -\frac{1}{2}x^2 \leq x^3 &\iff \\ -\frac{1}{2} \leq x\end{aligned}$$

which holds because $\frac{1}{2} < 1+x$. Note that putting $n = 0$ would have required us to prove $0 \leq x$, which fails.

This derivation reveals some limitations. We should have been able to prove this result with looser bounds on x, since $\ln(1 + x) \leq x$ holds for $x > -1$. We could not do this because our upper bound, $\overline{\ln}(x, n)$, introduces the value m in equation (16). This formulation allows the upper bound to

be tight, but for our purposes we need to seek looser bounds that have less restrictive range conditions.

The bounds for the exponential function have a similar problem. An alternative lower bound, valid for all $x \geq 0$, comes directly from its Taylor expansion:

$$\underline{\exp}(x, n) = \sum_{i=0}^{n} \frac{x^i}{i!}.$$

This series for the logarithm [1] also suggests a lower bound, for $x \geq 1$:

$$\underline{\ln}(x, n) = \sum_{i=1}^{n} \frac{(x-1)^i}{ix^i}.$$

Finding upper and lower bounds for elementary functions that work well with RCF decision procedures is one of our first tasks.

5 Conclusions

Our preliminary investigations are promising. We have used the method described above to solve the problems shown in Fig. 3. (Note that some of these split into several problems when the absolute value function is removed and chains of inequalities are separated.) We manually reduced each problem to algebraic form as described above, then tried to solve the reduced problems using three different tools.

- QEPCAD solved all of the problems, usually taking less than one second.
- HOL Light's sum-of-squares tool (`REAL_SOS`) solved all of the problems but two, again usually in less than a second.
- HOL Light's quantifier elimination tool (`REAL_QELIM_CONV`) solved all of the problems but three. It seldom required more than five seconds. The 351 seconds we reported above is clearly exceptional.

The simplest bound using $n = 0$ was sufficient for all but one of the problems, which required $n = 1$.

Much work remains to be done before this procedure can be automated. We need to experiment with a variety of upper and lower bounds. Case analyses will still be inevitable, so we need techniques to automate them in the most common situations. We need to tune the procedure by testing on a large suite of problems, and we have to evaluate different ways of deciding the RCF problems that are finally generated.

$$-\frac{1}{2} \le x \le 3 \Longrightarrow \frac{x}{1+x} \le \ln(1+x) \le x$$
$$-3 \le x \le \frac{1}{2} \Longrightarrow \frac{-x}{1-x} \le \ln(1-x) \le -x$$
$$0 \le x \le 3 \Longrightarrow |\ln(1+x) - x| \le x^2$$
$$-3 \le x \le 0 \Longrightarrow |\ln(1-x) + x| \le x^2$$
$$|x| \le \frac{1}{2} \Longrightarrow |\ln(1+x) - x| \le 2x^2$$
$$|x| \le \frac{1}{2} \Longrightarrow |\ln(1-x) + x| \le 2x^2$$
$$0 \le x \le 0.5828 \Longrightarrow |\ln(1-x)| \le \frac{3x}{2}$$
$$-0.5828 \le x \le 0 \Longrightarrow |\ln(1+x)| \le -\frac{3x}{2}$$
$$\frac{1}{2} \le x \le 4 \Longrightarrow \ln x \le x - 1$$
$$0 \le x \le 1 \Longrightarrow e^{(x-x^2)} \le 1 + x$$
$$-1 \le x \le 1 \Longrightarrow 1 + x \le e^x$$
$$-1 \le x \le 1 \Longrightarrow 1 - x \le e^{-x}$$
$$-1 \le x \le 1 \Longrightarrow e^x \le \frac{1}{1-x}$$
$$-1 \le x \le 1 \Longrightarrow e^{-x} \le \frac{1}{1+x}$$
$$x \le \frac{1}{2} \Longrightarrow e^{-x/(1-x)} \le 1 - x$$
$$-\frac{1}{2} \le x \Longrightarrow e^{x/(1+x)} \le 1 + x$$
$$0 \le x \le 1 \Longrightarrow e^{-x} \le 1 - \frac{x}{2}$$
$$-1 \le x \le 0 \Longrightarrow e^x \le 1 + \frac{x}{2}$$
$$0 \le |x| \le 1 \Longrightarrow \frac{1}{4}|x| \le |e^x - 1| \le \frac{7}{4}|x|$$

Figure 3: Problems Solved

Acknowledgements

The research was funded by the EPSRC grant EP/C013409/1, *Beyond Linear Arithmetic: Automatic Proof Procedures for the Reals*. R. W. Butler, D. Lester, J. Harrison and C. Muñoz answered many questions. A. Chaieb and the referees commented on this paper.

References

[1] M. Abramowitz and I. A. Stegun. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. Wiley, 1972.

[2] J. Avigad, K. Donnelly, D. Gray, and P. Raff. A formally verified proof of the prime number theorem. *ACM Trans. Comput. Logic*, in press.

[3] A. Dolzmann, T. Sturm, and V. Weispfenning. Real quantifier elimination in practice. Technical Report MIP-9720, Universität Passau, D-94030, Germany, 1997.

[4] J. Harrison. A HOL theory of Euclidean space. In Hurd and Melham [7], pages 114–129.

[5] H. Hong. QEPCAD — quantifier elimination by partial cylindrical algebraic decomposition. Available on the Internet at `http://www.cs.usna.edu/~qepcad/B/QEPCAD.html`.

[6] W. A. Hunt, Jr., R. B. Krug, and J. Moore. Linear and nonlinear arithmetic in ACL2. In D. Geist and E. Tronci, editors, *Correct Hardware Design and Verification Methods (CHARME)*, LNCS 2860, pages 319–333, 2003.

[7] J. Hurd and T. Melham, editors. *Theorem Proving in Higher Order Logics: TPHOLs 2005*, LNCS 3603. Springer, 2005.

[8] S. McLaughlin and J. Harrison. A proof-producing decision procedure for real arithmetic. In R. Nieuwenhuis, editor, *Automated Deduction — CADE-20 International Conference*, LNAI 3632, pages 295–314. Springer, 2005.

[9] C. Muñoz and D. Lester. Real number calculations and theorem proving. In Hurd and Melham [7], pages 195–210.

[10] P. A. Parrilo. Semidefinite programming relaxations for semialgebraic problems. *Mathematical Programming*, 96(2):293–320, 2003.

[11] D. Richardson. Some undecidable problems involving elementary functions of a real variable. *Journal of Symbolic Logic*, 33(4):514–520, Dec. 1968.

[12] A. Tiwari. Abstraction based theorem proving: An example from the theory of reals. In C. Tinelli and S. Ranise, editors, *PDPAR: Workshop on Pragmatics of Decision Procedures in Automated Deduction*, pages 40–52. INRIA, Nancy, 2003.

Rewrite-Based Satisfiability Procedures for Recursive Data Structures

Maria Paola Bonacina and **Mnacho Echenim**
Dipartimento di Informatica
Università degli Studi di Verona, Italy

Abstract

If a rewrite-based inference system is guaranteed to terminate on the axioms of a theory $\mathcal{T}$ and any set of ground literals, then any theorem-proving strategy based on that inference system is a rewrite-based decision procedure for $\mathcal{T}$-satisfiability. In this paper, we consider the class of theories defining *recursive data structures*, that might appear out of reach for this approach, because they are defined by an infinite set of axioms. We overcome this obstacle by designing a problem reduction that allows us to prove a general termination result for all these theories. We also show that the theorem-proving strategy decides satisfiability problems in any combination of these theories with other theories decided by the rewrite-based approach.

1 Introduction

Most state-of-the-art verification tools rely on built-in satisfiability procedures for specific theories. These satisfiability procedures can be quite complicated to design and combine, and significant effort is devoted to proving them correct and complete, and implementing them. A new approach to defining satisfiability procedures was introduced in [3], where the authors showed that a sound and complete first-order theorem-proving strategy can be used to solve satisfiability problems for several theories of data structures. The idea behind this approach is that since such a strategy is a semi-decision procedure for first-order validity, if one proves that it *terminates* on a presentation of the theory of interest $\mathcal{T}$ and any set of ground literals, then it is a decision procedure for $\mathcal{T}$*-satisfiability*. In [3], this idea was applied to a standard inference system, the superposition calculus $\mathcal{SP}$, and several theories, including those of *arrays* and *possibly cyclic lists*.

Since most verification problems involve more than one theory, a significant advantage of an approach based on generic reasoning is that it makes it conceptually simple to combine theories, by considering the union of their presentations. Along with several experimental results that showed the practicality of the rewrite-based approach, the authors of [1] defined the notion

of *variable-inactive theories*. This variable-inactivity condition guarantees that $\mathcal{SP}$ terminates on a combination of theories, provided it terminates on each individual theory. The authors showed that an $\mathcal{SP}$-based strategy is a satisfiability procedure for any combination of the theories of [3] and those they considered.

Several of the theories for which $\mathcal{SP}$ has been shown to yield satisfiability procedures involve *lists*. The superposition calculus yields satisfiability procedures for the theories of lists *à la* Shostak and *à la* Nelson and Oppen (see [3]), and for the theory of lists with nil (see [2]). A theory of lists that was not yet considered is that of *acyclic lists*, where formulae such as $\mathrm{car}(x) \simeq x$ are unsatisfiable. This theory, along with that of *integer offsets* considered in [6, 1], belong to the general class of *theories of recursive data structures*, that we denote $\mathcal{RDS}$. Each member of this class is denoted $\mathcal{RDS}_k$, where k represents the number of *selectors* in the theory. We shall see that the theory of integer offsets is $\mathcal{RDS}_1$, and the theory of acyclic lists is $\mathcal{RDS}_2$. In this paper, we investigate how a rewrite-based inference system can be used to solve any $\mathcal{RDS}_k$-satisfiability problem, for any k. The contributions of the paper are the following:

- Every theory in the class $\mathcal{RDS}$ is presented by an infinite set of axioms, which cannot be given as an input to a theorem prover. Here, we present a reduction that conquers this infinite presentation problem.
- We prove that for any fair search plan, the inference system terminates on any reduced $\mathcal{RDS}_k$-satisfiability problem.
- We show that for every k, the theory $\mathcal{RDS}_k$ can be combined with all the theories considered in [3, 1, 2], namely those of lists *à la* Shostak and *à la* Nelson and Oppen, arrays and records with or without extensionality, sets with extensionality, possibly empty lists and integer offsets modulo.

Related work. Theories of recursive data structures were studied by Oppen in [8], where he described a linear satisfiability procedure for the case where uninterpreted function symbols are excluded. In [9], Zhang et al. investigated quantifier-elimination problems for an extension of the theory considered by Oppen: their setting includes atoms (constants) and several different constructors. However, their setting also excludes uninterpreted function symbols. They provided a satisfiability procedure for this theory that "guesses" a so-called *type completion*, to determine which constructor was used on each term, or whether the term is an atom, and then calls Oppen's algorithm.

In this paper, we consider the recursive data structures as defined in [8], since our aim was to investigate how to apply the rewrite-based methodology to theories defined by *infinite sets of axioms*. Similar to any other

theory for which the superposition calculus can be used as a satisfiability procedure, all these theories can be combined with the theory of equality with uninterpreted functions. For instance, it can be used to prove the $\mathcal{RDS}_k$-unsatisfiability of a set such as

$$S \;=\; \{\mathsf{cons}(c_1,\ldots,c_k) \simeq c,\ \mathsf{cons}(c_1,\ldots,c_k) \simeq c',\ f(c) \not\simeq f(c')\},$$

where f is an uninterpreted function symbol.

Due to space restrictions, the proofs were not included in this paper. They can all be found in [5].

Preliminaries

In the following, given a signature Σ, we consider the standard definitions of Σ-terms, Σ-literals and Σ-theories. The symbol $\simeq$ denotes unordered equality, and $\bowtie$ is either $\simeq$ or $\not\simeq$. Unless stated otherwise, the letters x and y will denote variables, d and e elements of an interpretation domain, and all other lower-case letters will be constants or function symbols in Σ. Given a term t, $\mathrm{Var}(t)$ denotes the set of variables appearing in t. If t is a constant or a variable, then the *depth* of t is $\mathrm{depth}(t) = 0$, and otherwise, $\mathrm{depth}(f(t_1,\ldots,t_n)) = 1 + \max\{\mathrm{depth}(t_i) \mid i = 1,\ldots,n\}$. The *depth* of a literal is defined by $\mathrm{depth}(l \bowtie r) = \mathrm{depth}(l) + \mathrm{depth}(r)$. A positive literal is *flat* if its depth is 0 or 1, and a negative literal is *flat* if its depth is 0. We will make use of the following standard result: given a signature Σ and a Σ-theory $\mathcal{T}$, let S be a finite set of Σ-literals. Then there exists a signature Σ' obtained from Σ by adding a finite number of constants, and a finite set S' of flat Σ'-literals such that S' is $\mathcal{T}$-satisfiable if and only if S is.

A *simplification ordering* $\succ$ is an ordering that is *stable*, *monotonic* and contains the *subterm ordering*: if $s \succ t$, then $c[s]\sigma \succ c[t]\sigma$ for any context c and substitution σ, and if t is a subterm of s then $s \succ t$. A *complete simplification ordering*, or CSO, is a simplification ordering that is total on ground terms. We write $t \prec s$ if and only if $s \succ t$. More details on orderings can be found, e.g., in [4]. A CSO is extended to literals and clauses by multiset extension as usual, and when no confusion is possible, we will mention maximal literals without any reference to $\succ$.

The *superposition calculus*, or $\mathcal{SP}$, is a *rewrite-based inference system* which is refutationally complete for first-order logic with equality (see, e.g., [7]). It consists of *expansion* and *contraction rules*, and is based on a CSO on terms which is extended to literals and clauses in a standard way. Given a CSO $\succ$, we write $\mathcal{SP}_\succ$ for $\mathcal{SP}$ with $\succ$. An $\mathcal{SP}_\succ$*-derivation* is a sequence

$$S_0 \vdash_{\mathcal{SP}_\succ} S_1 \vdash_{\mathcal{SP}_\succ} \ldots S_i \vdash_{\mathcal{SP}_\succ} \ldots,$$

each S_i being a set of clauses obtained by applying an expansion or a contraction rule to clauses in S_{i-1}. Such a derivation yields a set of *persistent*

clauses:

$$S_\infty \;=\; \bigcup_{j\geq 0}\bigcap_{i\geq j} S_i,$$

which can of course be infinite. Given a finite set of ground literals S, in order to prove that the set of persistent clauses obtained by a fair $\mathcal{SP}_\succ$-derivation from $\mathcal{T}\cup S$ is finite, we may impose additional restrictions on the CSO $\succ$. Any CSO verifying these restrictions will be termed as *$\mathcal{T}$-good*. We also say that an $\mathcal{SP}_\succ$-strategy is *$\mathcal{T}$-good* if the CSO $\succ$ is $\mathcal{T}$-good.

A clause C is *variable-inactive for* $\succ$ if no maximal literal in C is an equation $t \simeq x$, where $x \notin \mathrm{Var}(t)$. A set of clauses is *variable-inactive for* $\succ$ if all its clauses are variable-inactive for $\succ$. A theory presentation $\mathcal{T}$ is *variable-inactive for* $\succ$ if the limit S_∞ of any fair $\mathcal{SP}_\succ$-derivation from $S_0 = \mathcal{T}\cup S$ is variable-inactive. When no confusion is possible, we will say that a clause (resp. a set of clauses or a theory presentation) is variable-inactive, without any mention of $\succ$.

2 The theory of recursive data structures

The theory $\mathcal{RDS}_k$ of recursive data structures is based on the following signature:

$$\begin{aligned}\Sigma_{\mathcal{RDS}_k} &= \{\mathsf{cons}\}\cup\Sigma_{sel},\\ \Sigma_{sel} &= \{\mathsf{sel}_1,\ldots,\mathsf{sel}_k\},\end{aligned}$$

where cons has arity k, and the sel_i's all have arity 1. The function symbols $\mathsf{sel}_1,\ldots,\mathsf{sel}_k$ stand for the *selectors*, and cons stands for the *constructor*. This theory is axiomatized by the following (infinite) set of axioms, denoted $Ax(\mathcal{RDS}_k)$:

$$\begin{aligned}\mathsf{sel}_i(\mathsf{cons}(x_1,\ldots,x_i,\ldots,x_k)) &\simeq x_i \quad \text{for } i=1,\ldots,k\\ \mathsf{cons}(\mathsf{sel}_1(x),\ldots,\mathsf{sel}_k(x)) &\simeq x,\\ t[x] &\not\simeq x,\end{aligned}$$

where x and the x_i's are (implicitly) universally quantified variables and $t[x]$ is any compound Σ_{sel}-term where the variable x occurs. The axioms $t[x]\not\simeq x$ are *acyclicity* axioms that prevent the theory from entailing equations such as $\mathsf{sel}_1(\mathsf{sel}_2(\mathsf{sel}_3(x)))\simeq x$.

For the sake of clarity, we also define

$$\begin{aligned}Ac &= \{t[x]\not\simeq x \mid t[x] \text{ is a } \Sigma_{sel}\text{-term}\},\\ Ac[n] &= \{t[x]\not\simeq x \mid t[x] \text{ is a } \Sigma_{sel}\text{-term and } \mathrm{depth}(t[x])\leq n\}.\end{aligned}$$

Example 1 Consider the case where $k = 2$. If we write $\mathrm{car}(x)$ instead of $\mathsf{sel}_1(x)$ and $\mathrm{cdr}(x)$ instead of $\mathsf{sel}_2(x)$, then our axioms become:

$$\begin{array}{rcl} \mathrm{car}(\mathsf{cons}(x,y)) & \simeq & x, \\ \mathrm{cdr}(\mathsf{cons}(x,y)) & \simeq & y, \\ \mathsf{cons}(\mathrm{car}(x),\mathrm{cdr}(x)) & \simeq & x, \\ t[x] & \not\simeq & x, \end{array}$$

and for example, we have:

$$\begin{array}{rll} Ac[2] = & \{\mathrm{car}(\mathrm{car}(x)) \not\simeq x, & \mathrm{cdr}(\mathrm{cdr}(x)) \not\simeq x, \\ & \mathrm{car}(\mathrm{cdr}(x)) \not\simeq x, & \mathrm{cdr}(\mathrm{car}(x)) \not\simeq x\}. \end{array}$$

We consider the problem of checking the $\mathcal{RDS}_k$-satisfiability of a set S of ground (equational) literals built out of the symbols in $\Sigma_{\mathcal{RDS}_k}$ and a set of finitely many constant symbols. This is done by checking the satisfiability of the following set of clauses:

$$Ax(\mathcal{RDS}_k) \quad \cup \quad S.$$

According to the methodology of [3, 1, 2], this problem is solved in three phases:

Flattening: flatten all ground literals in the original problem, thus obtaining an equisatisfiable set of flat literals,

$\mathcal{RDS}_k$-reduction: transform the flattened problem into an equisatisfiable $\mathcal{RDS}_k$-*reduced* problem consisting of a *finite* set of clauses,

Termination: prove that any fair $\mathcal{SP}_\succ$-strategy terminates on the $\mathcal{RDS}_k$-reduced problems.

The flattening step is straightforward, and we now focus on the $\mathcal{RDS}_k$-reduction step.

3 $\mathcal{RDS}_k$-reduction

The aim of a reduction is to transform a formula into another one which is equisatisfiable and easier to work on. Here, given a formula S, we want to transform it into a formula which is equisatisfiable in a theory that does not axiomatize the relationship between the constructor and the selectors. We begin by observing that S can be transformed by suppressing either every occurrence of cons, or every occurrence of the sel_i's.

Example 2 Consider the case where $k = 2$, and let

$$S \quad = \quad \{\mathsf{cons}(c_1, c_2) \simeq c,\ \mathsf{sel}_1(c) \simeq c'_1\}.$$

If we remove the occurrence of cons, S would become

$$S_1 \quad = \quad \{\mathsf{sel}_1(c) \simeq c_1,\ \mathsf{sel}_2(c) \simeq c_2,\ \mathsf{sel}_1(c) \simeq c'_1\}.$$

If we remove the occurrence of sel_1, S would become

$$S_2 \quad = \quad \{\mathsf{cons}(c_1, c_2) \simeq c,\ c_1 \simeq c'_1\}.$$

We choose to remove every occurrence of cons because it is easier to work with function symbols of arity 1:

Definition 3 A set of ground flat literals is $\mathcal{RDS}_k$*-reduced* if it contains no occurrence of cons. ◇

Given a set S of ground flat literals, the symbol cons may appear only in literals of the form $\mathsf{cons}(c_1, \ldots, c_k) \simeq c$ for constants $c, c_1, \ldots, c_k$. Negative ground flat literals are of the form $c \not\simeq c'$ and therefore do not contain any occurrence of cons. The $\mathcal{RDS}_k$-reduction of S is obtained by replacing every literal $\mathsf{cons}(c_1, \ldots, c_k) \simeq c$ appearing in S by the literals $\mathsf{sel}_1(c) \simeq c_1, \ldots, \mathsf{sel}_k(c) \simeq c_k$. The resulting $\mathcal{RDS}_k$-reduced form S' of S is denoted $Red_{\mathcal{RDS}_k}(S)$, and it is obviously unique.

It is not intuitive in which theory the $\mathcal{RDS}_k$-reduced form of S is equisatisfiable to S, and we need the following definition:

Definition 4 Let (**ext**) denote the following "extensionality lemma":

$$\bigwedge_{i=1}^{k} (\mathsf{sel}_k(x) \simeq \mathsf{sel}_k(y)) \quad \Rightarrow \quad x \simeq y. \qquad \diamond$$

Proposition 5 *The extensionality lemma is logically entailed by the axiom* $\mathsf{cons}(\mathsf{sel}_1(x), \ldots, \mathsf{sel}_k(x)) \simeq x$.

We can then show that $\mathcal{RDS}_k$-reduction reduces satisfiability w.r.t. $Ax(\mathcal{RDS}_k)$ to satisfiability w.r.t. $Ac \cup \{(\mathbf{ext})\}$.

Lemma 6 *Let S be a set of ground flat literals, then $Ax(\mathcal{RDS}_k) \cup S$ is satisfiable if and only if $Ac \cup \{(\mathbf{ext})\} \cup Red_{\mathcal{RDS}_k}(S)$ is.*

The first implication of this lemma is quite simple to prove, we show on an example the idea behind the proof of the converse implication.

Example 7 Consider the case where $k = 1$ and $S = \{\mathsf{cons}(c') \simeq c\}$, the $\mathcal{RDS}_1$-reduced form of S is therefore $S' = \{\mathsf{sel}_1(c) \simeq c'\}$. We consider the model $M = (\mathbb{N}, I)$ of $Ac \cup \{(\mathbf{ext})\} \cup S'$, where I interprets c as 0, c' as 1, and sel_1 as the successor function on natural numbers. We construct a model $M' = (D', I')$ of $Ax(\mathcal{RDS}_1) \cup S$, such that $D \subseteq D'$ and I' is identical to I on D, as follows.

Intuitively, since M' must satisfy the axiom $\mathsf{cons}(\mathsf{sel}_1(x)) \simeq x$, I' will interpret cons as the predecessor function. Hence, for every $d \in \mathbb{N} \setminus \{0\}$, $\mathsf{cons}^{I'}(d)$ is the predecessor of d.

However, the predecessor function on natural numbers is not defined for 0. In order to define correctly the value of $\mathsf{cons}^{I'}(0)$, we augment D with a new element, say -1, and obtain a new set D_1. Then we define $\mathsf{cons}^{I'}(0) = -1$, and $\mathsf{sel}_1^{I'}(-1) = 0$. We now need to define the value of $\mathsf{cons}^{I'}(-1)$, so we augment D_1 with a new element, say -2, and define $\mathsf{cons}^{I'}(-1) = -2$, and $\mathsf{sel}_1^{I'}(-2) = -1$.

By iterating this process, we obtain the model $M' = (D', I')$, where $D' = \mathbb{Z}$, I' interprets sel_1 as the standard successor function on integers, and cons as the standard predecessor function on integers. It is clear that M' is a model of $Ax(\mathcal{RDS}_1) \cup S$.

It is also possible to define a notion of $\mathcal{RDS}_k$-reduction where every occurrence of the sel_i's is removed. However, no additional property is gained by using this other alternative, and the corresponding reduction is less intuitive.

4 From Ac to $Ac[n]$

The set Ac being infinite, $\mathcal{SP}$ cannot be used as a satisfiability procedure on any set of the form $Ac \cup \{(\mathbf{ext})\} \cup S$. Thus, the next move is to bound the number of axioms in Ac needed to solve the satisfiability problem, and try to consider an $Ac[n]$ instead of Ac. It is clear that for any n and any set S, a model of $Ac \cup \{(\mathbf{ext})\} \cup S$ is also a model of $Ac[n] \cup \{(\mathbf{ext})\} \cup S$, the difficulty is therefore to determine an n for which a model of $Ac \cup S$ is guaranteed to exist, provided $Ac[n] \cup \{(\mathbf{ext})\} \cup S$ is satisfiable. The following example provides the intuition that this bound depends on the number of selectors in S.

Example 8 Let $S = \{\mathsf{sel}_1(c_1) \simeq c_2, \mathsf{sel}_2(c_2) \simeq c_3, \mathsf{sel}_3(c_3) \simeq c_4, c_1 \simeq c_4\}$. Then:

$$\begin{array}{llll} Ac[1] \cup \{(\mathbf{ext})\} \cup S & \text{and} & Ac[2] \cup \{(\mathbf{ext})\} \cup S & \text{are satisfiable,} \\ Ac[3] \cup \{(\mathbf{ext})\} \cup S & \text{and} & Ac \cup \{(\mathbf{ext})\} \cup S & \text{are unsatisfiable.} \end{array}$$

The following lemma allows us prove that having n occurrences of selectors implies that it is indeed sufficient to consider $Ac[n]$ instead of Ac.

Lemma 9 *Let S be an $\mathcal{RDS}_k$-reduced set of ground flat literals and let l be the number of occurrences of selectors in S. For $n \geq l$, suppose that $Ac[n] \cup \{(\mathbf{ext})\} \cup S$ is satisfiable. Then $Ac[n+1] \cup \{(\mathbf{ext})\} \cup S$ is also satisfiable.*

A simple induction using Lemma 9 shows that for every $k \geq 0$, if $Ac[n] \cup S$ is satisfiable, then so is $Ac[n+k] \cup S$. We can therefore deduce:

Corollary 10 *Let S be an $\mathcal{RDS}_k$-reduced set of ground flat literals and let n be the number of occurrences of selectors in S. Then, $Ac \cup \{(\mathbf{ext})\} \cup S$ is satisfiable if and only if $Ac[n] \cup \{(\mathbf{ext})\} \cup S$ is.*

5 $\mathcal{SP}_\succ$ as a satisfiability procedure

We now show that only a finite number of clauses are generated by the superposition calculus on any set $Ac[n] \cup \{(\mathbf{ext})\} \cup S$, where S is $\mathcal{RDS}_k$-reduced. This will be the case provided we use an $\mathcal{RDS}_k$-good CSO:

Definition 11 A CSO $\succ$ is $\mathcal{RDS}_k$-good if $t \succ c$ for every ground compound term t and every constant c. $\diamond$

Lemma 12 *Let $S_0 = Ac[n] \cup \{(\mathbf{ext})\} \cup S$, where S is a finite $\mathcal{RDS}_k$-reduced set of ground flat literals. Consider the limit S_∞ of the derivation $S_0 \vdash_{\mathcal{SP}_\succ} S_1 \vdash_{\mathcal{SP}_\succ} \ldots$ generated by a fair $\mathcal{RDS}_k$-good $\mathcal{SP}_\succ$-strategy; every clause in S_∞ belongs to one of the categories enumerated below:*

i) the empty clause;

ii) the clauses in $Ac[n] \cup \{(\mathbf{ext})\}$, i.e.

  a) $t[x] \not\simeq x$, where t is a Σ_{sel}-term of depth at most n,

  b) $x \simeq y \ \vee \left(\bigvee_{i=1}^{k} (\mathsf{sel}_i(x) \not\simeq \mathsf{sel}_i(y))\right)$;

iii) ground clauses of the form

  a) $c \simeq c' \vee (\bigvee_{j=1}^{m} d_j \not\simeq d'_j)$ where $m \geq 0$,

  b) $f(c) \simeq c' \vee (\bigvee_{j=1}^{m} d_j \not\simeq d'_j)$ where $m \geq 0$,

  c) $t[c] \not\simeq c' \vee (\bigvee_{j=1}^{m} d_j \not\simeq d'_j)$, where t is a compound Σ_{sel}-term of depth at most $n-1$ and $m \geq 0$,

  d) $\bigvee_{j=1}^{m} d_j \not\simeq d'_j$, where $m \geq 1$;

iv) clauses of the form

$$c \simeq x \vee \left(\bigvee_{p=1}^{j} \mathsf{sel}_{i_p}(c) \not\simeq \mathsf{sel}_{i_p}(x)\right) \ \vee \ \left(\bigvee_{p=j+1}^{k} c_{i_p} \not\simeq \mathsf{sel}_{i_p}(x)\right) \ \vee \ \left(\bigvee_{j=1}^{m} d_j \not\simeq d'_j\right)$$

where $i_1, \ldots, i_k$ *is a permutation of* $1, \ldots, k$, $0 \leq j \leq k$ *and* $m \geq 0$;

v) clauses of the form

$$c \simeq c' \vee \left(\bigvee_{p=1}^{j_1}(\mathsf{sel}_{i_p}(c) \not\simeq \mathsf{sel}_{i_p}(c'))\right) \vee \left(\bigvee_{p=j_1+1}^{j_2}(\mathsf{sel}_{i_p}(c) \not\simeq c'_{i_p})\right) \vee \left(\bigvee_{p=j_2+1}^{j_3}(c_{i_p} \not\simeq \mathsf{sel}_{i_p}(c'))\right) \vee \left(\bigvee_{p=j_3+1}^{k}(c_{i_p} \not\simeq c'_{i_p})\right) \vee \left(\bigvee_{j=1}^{m} d_j \not\simeq d'_j\right)$$

where $i_1, \ldots, i_k$ *is a permutation of* $1, \ldots, k$, $0 \leq j_1 \leq j_2 \leq j_3 \leq k$, $j_3 > 0$ *and* $m \geq 0$.

This lemma is proved by induction on the length l of the derivations. The result is true for $l = 0$: the clauses in S_0 are in *(ii)* or *(iii)* with $m = 0$. A case analysis shows that if the result is true for $l - 1$, then it is also true for l. We give an example of such a derivation:

Example 13 Consider the case where $k = 3$, and suppose we want to test the unsatisfiability of the following set:

$$\begin{array}{lrcllcllcll} S = \{ & \mathsf{sel}_1(c) & \simeq & d_1, & \mathsf{sel}_2(c') & \simeq & d'_2, & \mathsf{sel}_2(c) & \simeq & d_2, & \\ & \mathsf{sel}_1(c') & \simeq & d'_1, & \mathsf{sel}_3(c) & \simeq & d_3, & \mathsf{sel}_3(c') & \simeq & d'_3, & \\ & d_1 & \simeq & d'_1, & d_2 & \simeq & d'_2, & d_3 & \simeq & d'_3, & \\ & & & & c & \not\simeq & c' & & & & \}. \end{array}$$

- A superposition of $\mathsf{sel}_1(c) \simeq d_1$ into $\{(\mathbf{ext})\}$ yields a clause in *(iv)* (with $m = 0$):

$$c \simeq x \vee \Big(\underline{\mathsf{sel}_2(c) \not\simeq \mathsf{sel}_2(x)} \vee \mathsf{sel}_3(c) \not\simeq \mathsf{sel}_3(x)\Big) \vee \Big(d_1 \not\simeq \mathsf{sel}_1(x)\Big),$$

- A superposition of $\mathsf{sel}_2(c') \simeq d'_2$ into the underlined literal of this clause yields a clause in *(v)*:

$$c \simeq c' \vee \Big(\mathsf{sel}_3(c) \not\simeq \mathsf{sel}_3(c')\Big) \vee \Big(\underline{\mathsf{sel}_2(c) \not\simeq d'_2}\Big) \vee \Big(d_1 \not\simeq \mathsf{sel}_1(c')\Big),$$

- A simplification of this clause by $\mathsf{sel}_2(c) \simeq d_2$ yields a clause in *(v)*:

$$c \simeq c' \vee \Big(\mathsf{sel}_3(c) \not\simeq \mathsf{sel}_3(c')\Big) \vee \Big(d_1 \not\simeq \mathsf{sel}_1(c')\Big) \vee \Big(d_2 \not\simeq d'_2\Big),$$

- Further simplifications by $\mathsf{sel}_1(c') \simeq d'_1$, $\mathsf{sel}_3(c) \simeq d_3$ and $\mathsf{sel}_3(c') \simeq d'_3$ yield the clause

$$c \simeq c' \vee \Big(\bigvee_{i=1}^{3} d_i \not\simeq d'_i\Big).$$

- The simplifications by $d_i \simeq d'_i$ for $i = 1, \ldots, 3$ yield the clause $c \simeq c'$, which together with $c \not\simeq c'$ produces the empty clause.

Since the signature is finite, there are finitely many clauses such as those enumerated in Lemma 12. We therefore deduce:

Corollary 14 *Any fair $\mathcal{RDS}_k$-good $\mathcal{SP}_\succ$-strategy terminates when applied to $Ac[n] \cup \{(\mathbf{ext})\} \cup S$, where S is a finite $\mathcal{RDS}_k$-reduced set of ground flat literals.*

We can also evaluate the complexity of this procedure by determining the number of clauses in each of the categories defined in Lemma 12.

Theorem 15 *Any fair $\mathcal{RDS}_k$-good $\mathcal{SP}_\succ$-strategy is an exponential satisfiability procedure for $\mathcal{RDS}_k$.*

PROOF. Let n be the number of literals in S, both the number of constants and the number of selectors appearing in S are therefore in $O(n)$. We examine the cardinalities of each of the categories defined in Lemma 12.

- Category *(ii)* contains $O(n)$ clauses if $k = 1$ and $O(k^n)$ clauses if $k \geq 2$.
- Clauses in categories *(iii)*, *(iv)* or *(v)* can contain any literal of the form $d \not\simeq d'$ where d and d' are constants, thus, these categories all contain $O(2^{n^2})$ clauses.

Hence, the total number of clauses generated is bound by a constant which is $O(2^{n^2})$, and since each inference step is polynomial, the overall procedure is in $O(2^{n^2})$. ∎

Although this complexity bound is exponential, it measures the size of the saturated set. Since a theorem prover seeks to generate a proof, as opposed to a saturated set, the relevance of this result with respect to predicting the performance of a theorem prover can therefore be quite limited.

One could actually have expected this procedure to be exponential for $k \geq 2$, since in that case $Ac[n]$ contains an exponential number of axioms. However the procedure is also exponential when $k = 1$, and a more careful analysis shows that this complexity is a consequence of the presence of (**ext**). In fact, it was shown in [2] that any fair $\mathcal{SP}_\succ$-strategy is a polynomial satisfiability procedure for the theory presented by the set of acyclicity axioms Ac when $k = 1$.

We finally address combination by proving that $\mathcal{RDS}_k$ is variable-inactive for $\mathcal{SP}_\succ$.

Theorem 16 *Let $S_0 = Ac[n] \cup S \cup \{(\mathbf{ext})\}$, where S is an $\mathcal{RDS}_k$-reduced set of ground flat literals, and n is the number of occurrences of selectors in S. Then S_∞ is variable-inactive.*

PROOF. The clauses in S_∞ belong to one of the classes enumerated in Lemma 12. Thus, the only clauses of S_∞ that may contain a literal $t \simeq x$ where $x \notin \mathrm{Var}(t)$ are in class *(iv)*. Since $\succ$ is a CSO, the literals $t \simeq x$ cannot be maximal in those clauses. ∎

This shows that the rewrite-based approach to satisfiability procedures can be applied to the combination of $\mathcal{RDS}_k$ with any number of the theories considered in [3, 1], including those of arrays and records with or without extensionality.

6 Conclusion

In this paper, we considered a class of theories representing recursive data structures, each of which is defined by an infinite set of axioms. We showed that the superposition calculus can be used as the basis of a satisfiability procedure for any theory in this class, and this result was obtained by defining a reduction that permits to restrict the number of acyclicity axioms to be taken into account.

A main issue we plan to investigate is complexity, since the basic procedure is exponential. A linear algorithm for such structures was obtained in [8], but it excludes uninterpreted function symbols. The setting of [6] includes uninterpreted function symbols, but the authors gave a polynomial algorithm only for the case where $k = 1$ (the theory of integer offsets). We intend to investigate making the complexity of the rewrite-based procedure dependent on k, and improving the bound for $k = 1$.

From the point of view of practical efficiency, we plan to test the performance of a state-of-the-art theorem prover on problems featuring this theory, possibly combined with those of [3, 1], and compare it with systems implementing decision procedures from other approaches. In this context, we may work on designing specialized search plans for satisfiability problems. Another direction for future work is to examine how the rewrite-based approach applies to recursive data structures with an atom predicate, or multiple constructors.

Acknowledgments

The authors wish to thank Silvio Ranise for bringing this class of theories to their attention and for his encouragement.

References

[1] Alessandro Armando, Maria Paola Bonacina, Silvio Ranise, and Stephan Schulz. On a rewriting approach to satisfiability procedures: Extension, combination of theories and an experimental appraisal. In Bernhard

Gramlich, editor, *Frontiers of Combining Systems, 5th International Workshop, FroCoS 2005, Vienna, Austria, September 19-21, 2005, Proceedings*, volume 3717 of *Lecture Notes in Computer Science*, pages 65–80. Springer, 2005.

[2] Alessandro Armando, Maria Paola Bonacina, Silvio Ranise, and Stephan Schulz. On a rewriting approach to satisfiability procedures: Theories of data structures, modularity and experimental appraisal. Technical Report RR 36/2005, Dipartimento di Informatica, Università degli Studi di Verona, 2006.

[3] Alessandro Armando, Silvio Ranise, and Michaël Rusinowitch. A rewriting approach to satisfiability procedures. *Inf. Comput.*, 183(2):140–164, 2003.

[4] Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[5] Maria Paola Bonacina and Mnacho Echenim. Generic theorem proving for decision procedures. Technical report, Università degli studi di Verona, 2006. Available at `http:profs.sci.univr.it/~echenim/`.

[6] Robert Nieuwenhuis and Albert Oliveras. Congruence closure with integer offsets. In Moshe Y. Vardi and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 10th International Conference, LPAR 2003, Almaty, Kazakhstan, September 22-26, 2003, Proceedings*, volume 2850 of *Lecture Notes in Computer Science*, pages 78–90. Springer, 2003.

[7] Robert Nieuwenhuis and Albert Rubio. Paramodulation-based theorem proving. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 371–443. Elsevier and MIT Press, 2001.

[8] Derek C. Oppen. Reasoning about recursively defined data structures. *J. ACM*, 27(3):403–411, 1980.

[9] Ting Zhang, Henny B. Sipma, and Zohar Manna. Decision procedures for recursive data structures with integer constraints. In David A. Basin and Michaël Rusinowitch, editors, *Automated Reasoning - Second International Joint Conference, IJCAR 2004, Cork, Ireland, July 4-8, 2004, Proceedings*, volume 3097 of *Lecture Notes in Computer Science*, pages 152–167. Springer, 2004.

An Abstract Decision Procedure for Satisfiability in the Theory of Recursive Data Types

Clark Barrett[1] Igor Shikanian[1] Cesare Tinelli[2]
[1]New York University, barrett|igor@cs.nyu.edu
[2]The University of Iowa, tinelli@cs.uiowa.edu

Abstract

The theory of recursive data types is a valuable modeling tool for software verification. In the past, decision procedures have been proposed for both the full theory and its universal fragment. However, previous work has been limited in various ways. In this paper, we present a general algorithm for the universal fragment. The algorithm is presented declaratively as a set of abstract rules which are terminating, sound, and complete. We show how other algorithms can be realized as strategies within our general framework. Finally, we propose a new strategy and give experimental results showing that it performs well in practice.

1 Introduction

Recursive data types are commonly used in programming. The same notion is also a convenient abstraction for common data types such as records and data structures such as linked lists used in more conventional programming languages. The ability to reason automatically and efficiently about recursive data types thus provides an important tool for the analysis and verification of programs.

Perhaps the best-known example of a simple recursive data type is the *list* type used in LISP. Lists are either the *null* list or are constructed from other lists using the *constructor cons*. This constructor takes two arguments and returns the result of prepending its first argument to the list in its second argument. In order to retrieve the elements of a list, a pair of *selectors* is provided: *car* returns the first element of a list and *cdr* returns the rest of the list.

More generally, we are interested in any set of (possibly mutually) recursive data types, each of which contains one or more constructors. Each constructor has selectors that can be used to retrieve the original arguments as well as a *tester* which indicates whether a given term was constructed using that constructor. As an example of the more general case, suppose we want to model lists of trees of natural numbers. Consider a set of three recursive data types: *nat*, *list*, and *tree*. *nat* has two constructors: *zero*, which takes no arguments (we call such a constructor a *nullary* constructor or *constant*); and *succ*, which takes a single argument of type *nat* and has the corresponding selector *pred*. The *list* type is as before except that we now specify that the elements of the list are of type *tree*. The *tree* type in turn has two constructors: *node*, which takes an argument of type *list* and has the corresponding selector *children*, and *leaf*, which takes an argument of type *nat* and has the corresponding selector *data*. We can represent this set of types using the following convenient notation based on that used in functional programming languages:

$$
\begin{array}{lcl}
\mathit{nat} & := & \mathit{succ}(\mathit{pred} : \mathit{nat}) \mid \mathit{zero}; \\
\mathit{list} & := & \mathit{cons}(\mathit{car} : \mathit{tree},\ \mathit{cdr} : \mathit{list}) \mid \mathit{null}; \\
\mathit{tree} & := & \mathit{node}(\mathit{children} : \mathit{list}) \mid \mathit{leaf}(\mathit{data} : \mathit{nat});
\end{array}
$$

The testers for this set of data types are *is_succ*, *is_zero*, *is_cons*, *is_null*, *is_node*, and *is_leaf*.

Propositions about a set of recursive data types can be captured in a sorted first-order language which closely resembles the structure of the data types themselves in that it has function symbols for each constructor and selector, and a predicate symbol for each tester. For instance, propositions that we would expect to be true for the example above include: (i) $\forall\, x : \mathit{nat}.\ \mathit{succ}(x) \neq \mathit{zero}$; (ii) $\forall\, x : \mathit{list}.\ x = \mathit{null} \vee \mathit{is\_cons}(x)$; and (iii) $\forall\, x : \mathit{tree}.\ \mathit{is\_leaf}(x) \rightarrow (\mathit{data}(x) = \mathit{zero} \vee \mathit{is\_succ}(\mathit{data}(x)))$.

In this paper, we discuss a procedure for deciding such formulas. We focus on satisfiability of a set of literals, which (through well-known reductions) can be used to decide the validity of universal formulas.

There are three main contributions of this work over earlier work on the topic. First, our setting is more general: we allow mutually recursive types and multiple constructors. The second contribution is in presentation. We present the theory itself in terms of an initial model rather than axiomatically as is often done. Also, the presentation of the decision procedure is given as abstract rewrite rules, making it more flexible and easier to analyze than if it were given imperatively. Finally, as described in Section 4, the flexibility provided by the abstract algorithm allows us to describe a new strategy with significantly improved practical efficiency.

Related Work. Term algebras over constructors provide the natural intended model for recursive data types. In [7] two dual axiomatizations of term algebras are presented, one with constructors only, the other with selectors and testers only.

An often-cited reference for the quantifier-free case is the treatment by Nelson and Oppen in 1980[11, 12] (where the problem is also shown to be NP-complete). In particular, Oppen's algorithm in [12] gives a detailed decision procedure for a single recursive data type with a single constructor; however, the case of multiple constructors is discussed only briefly and not rigorously.

More recently, several papers by Zhang et al. [14, 15] explore decision procedures for a single recursive data type. These papers focus on ambitious schemes for quantifier elimination and combinations with other theories. A possible extension of Oppen's algorithm to the case of multiple constructors is discussed briefly in [14]. A comparison of our algorithm with that of [14] is made in Section 4.

Finally, another approach based on first-order reasoning with the superposition calculus is described in [5]. This work shows how a decision procedure for a recursive data type can be automatically inferred from the first-order axioms, even though the axiomatization is infinite. Although the results are impressive from a theoretical point of view, the scope is limited to theories with a single constructor and the practical efficiency of such a scheme has yet to be shown.

2 The Theory of Recursive Data Types

Previous work on recursive data types (RDTs) uses first-order axiomatizations in an attempt to capture the main properties of a recursive data type and reason about it. We find it simpler and cleaner to use a semantic approach instead, as is done in algebraic

specification. A set of RDTs can be given a simple equational specification over a suitable signature. The intended model for our theory can be formally, and uniquely, defined as the initial model of this specification. Reasoning about a set of RDTs then amounts to reasoning about formulas that are true in this particular initial model.

2.1 Specifying RDTs

We formalize RDTs in the context of many-sorted equational logic (see [9] among others). We will assume that the reader is familiar with the basic notions in this logic, and also with basic notions of term rewriting.

We start with the theory signature. We assume a many-sorted signature Σ whose set of sorts consists of a distinguished sort bool for the Booleans, and $p \geq 1$ sorts $\tau_1, \ldots, \tau_p$ for the RDTs. We also allow $r \geq 0$ additional (non-RDT) sorts $\sigma_1, \ldots, \sigma_r$. We will denote by s, possibly with subscripts and superscripts, any sort in the signature other than bool, and by σ any sort in $\{\sigma_1, \ldots, \sigma_r\}$.

As mentioned earlier, the function symbols in our theory signature correspond to the constructors, selectors, and testers of the set of RDTs under consideration. We assume for each τ_i $(1 \leq i \leq p)$ a set of $m_i \geq 1$ *constructors* of τ_i. We denote these symbols as C^i_j, where j ranges from 1 to m_i. We denote the arity of C^i_j as n^i_j (0-arity constructors are also called nullary constructors or constants) and its sort as $s^i_{j,1} \times \cdots \times s^i_{j,n^i_j} \to \tau_i$. For each constructor C^i_j, we have a set of *selectors*, which we denote as $S^i_{j,k}$, where k ranges from 1 to n^i_j, of sort $\tau_i \to s^i_{j,k}$. Finally, for each constructor, there is a *tester*[1] $isC^i_j : \tau_i \to \mathsf{bool}$.

In addition to these symbols, we also assume that the signature contains two constants, true and false of sort bool, and an infinite number of distinct constants of each sort σ. The constants are meant to be names for the elements of that sort, so for instance if σ_1 were a sort for the natural numbers, we could use all the numerals as the constants of sort σ_1. Having all these constants in the signature is not necessary for our approach, but in the following exposition it provides an easy way of ensuring that the sorts in σ are infinite. Section 5.1 shows that our approach can be easily extended to the case in which some of these sorts are finite. To summarize, the set of function symbols of the signature Σ consists of:

$C^i_j : s^i_{j,1} \times \cdots \times s^i_{j,n^i_j} \to \tau_i$, for $i = 1, \ldots, p,\ j = 1, \ldots, m_i$,
$S^i_{j,k} : \tau_i \to s^i_{j,k}$, for $i = 1, \ldots, p,\ j = 1, \ldots, m_i,\ k = 1, \ldots, n^i_j$,
$isC^i_j : \tau_i \to \mathsf{bool}$, for $i = 1, \ldots, p,\ j = 1, \ldots, m_i$,
$\mathsf{true} : \mathsf{bool},\ \mathsf{false} : \mathsf{bool}$,
An infinite number of constants for each σ_l, for $l = 1, \ldots, r$.

As usual in many-sorted equational logic, we also have $p + r + 1$ equality symbols (one for each sort mentioned above), all written as $\approx$.

Our procedure requires one additional constraint on the set of RDTs: It must be *well-founded*. Informally, this means that each sort must contain terms that are not cyclic or infinite. More formally, we have the following definitions by simultaneous induction over constructors and sorts: (i) a constructor C^i_j is well-founded if all of its argument sorts are well-founded; (ii) the sorts $\sigma_1, \ldots, \sigma_r$ are all well-founded; (iii) a sort τ_i is well-founded if at least one of its constructors is well-founded. We require that every sort be well-founded according to the above definition.

[1] To simplify some of the proofs, and without loss of generality, we use functions to bool instead of predicates for the testers.

In some cases, it will be necessary to distinguish between *finite* and *infinite* τ-sorts: (i) a constructor is *finite* if it is nullary or if all of its argument sorts are finite; (ii) a sort τ_i is *finite* if all of its constructors are finite, and is *infinite* otherwise; (iii) the sorts $\sigma_1, \ldots, \sigma_r$ are all infinite. As we will see, consistent with the above terminology, our semantics will interpret finite, resp. infinite, τ-sorts indeed as finite, resp. infinite, sets.

We denote by $\mathcal{T}(\Sigma)$ the set of well-sorted ground terms of signature Σ or, equivalently, the (many-sorted) term algebra over that signature. The RDTs with functions and predicates denoted by the symbols of Σ are specified by the following set $\mathcal{E}$ of (universally quantified) equations. For reasons explained below, we assume that associated with every selector $S^i_{j,k} : \tau_i \to s^i_{j,k}$ is a distinguished ground term of sort $s^i_{j,k}$ containing no selectors (or testers), which we denote by $t^i_{j,k}$.

Equational Specification of the RDT: for $i = 1, \ldots, p$:

$$\forall x_1, \ldots, x_{n^i_j}.\ isC^i_j(C^i_j(x_1, \ldots, x_{n^i_j})) \approx \mathsf{true} \qquad (\text{for } j = 1, \ldots, m_i)$$

$$\forall x_1, \ldots, x_{n^i_{j'}}.\ isC^i_j(C^i_{j'}(x_1, \ldots, x_{n^i_{j'}})) \approx \mathsf{false} \qquad (\text{for } j, j' = 1, \ldots, m_i,\ j \neq j')$$

$$\forall x_1, \ldots, x_{n^i_j}.\ S^i_{j,k}(C^i_j(x_1, \ldots, x_{n^i_j})) \approx x_k \qquad (\text{for } k = 1, \ldots, n^i_j,\ j = 1, \ldots, m_i)$$

$$\forall x_1, \ldots, x_{n^i_{j'}}.\ S^i_{j,k}(C^i_{j'}(x_1, \ldots, x_{n^i_{j'}})) \approx t^i_{j,k} \qquad (\text{for } j, j' = 1, \ldots, m_i,\ j \neq j')$$

The last axiom specifies what happens when a selector is applied to the "wrong" constructor. Note that there is no obviously correct thing to do in this case since it would correspond to an error condition in a real application. Our axiom specifies that in this case, the result is the designated ground term for that selector. This is different from other treatments (such as [7, 14, 15]) where the application of a wrong selector is treated as the identity function. The main reason for this difference is that identity function would not always be well-sorted in multi-sorted logic.

By standard results in universal algebra we know that $\mathcal{E}$ admits an *initial model* $\mathcal{R}$ and we can show the following result:[2] Let Ω be the signature obtained from Σ by removing the selectors and the testers; then, the reduct of $\mathcal{R}$ to Ω is isomorphic to $\mathcal{T}(\Omega)$. Informally, this means that $\mathcal{R}$ does in fact capture the set of RDTs in question, as we can take the carrier of $\mathcal{R}$ to be the term algebra $\mathcal{T}(\Omega)$.

3 The Decision Procedure

In this section, we present a decision procedure for the satisfiability of sets of literals over $\mathcal{R}$. Our procedure builds on the algorithm by Oppen [12] for a single type with a single constructor. As an example of Oppen's algorithm, consider the *list* data type without the *null* constructor and the following set of literals: $\{cons(x, y) \approx z, car(w) \approx x, cdr(w) \approx y, w \not\approx z\}$. Oppen's algorithm uses a graph which relates terms according to their meaning in the intended model. Thus, $cons(x, y)$ is a parent of x and y and $car(w)$ and $cdr(w)$ are children of w. The equations induce an equivalence relation on the nodes of the graph. The Oppen algorithm proceeds by performing *upwards* (congruence) and *downwards* (unification) closure on the graph and then checking for cycles[3] or for a violation of any disequalities. For our example, upwards closure results in the conclusion $w \approx z$, which contradicts the disequality $w \not\approx z$.

[2] Proofs of all results in this paper can be found in [4].

[3] A simple example of a cycle is: $cdr(x) \approx x$.

As another example, consider the following set of literals: $\{cons(x,y) \approx z, car(w) \approx x, cdr(w) \approx y, v \approx w, y \not\approx cdr(v)\}$. The new graph has a node for v, with $cdr(v)$ as its right child. The Oppen algorithm requires that every node with at least one child have a complete set of children, so $car(v)$ is added as a left child of v. Now, downwards closure forces $car(v) \approx car(w) \approx x$ and $cdr(v) \approx cdr(w) \approx y$, contradicting the disequality $y \not\approx cdr(v)$.

An alternative algorithm for the case of a single constructor is to introduce new terms and variables to replace variables that are inside of selectors. For example, for the first set of literals above, we would introduce $w \approx cons(s,t)$ where s,t are new variables. Now, by substituting and collapsing applications of selectors to constructors, we get $\{cons(x,y) \approx z, w \approx cons(s,t), x \approx s, t \approx y, w \not\approx z\}$. In general, this approach only requires downwards closure.

Unfortunately, with the addition of more than one constructor, things are not quite as simple. In particular, the simple approach of replacing variables with constructor terms does not work because one cannot establish *a priori* whether the value denoted by a given variable is built with one constructor or another. A simple extension of Oppen's algorithm for the case of multiple constructors is proposed in [14]. The idea is to first guess a *type completion*, that is, a labeling of every variable by a constructor, which is meant to constrain a variable to take only values built with the associated constructor. Once all variables are labeled by a single constructor, the Oppen algorithm can be used to determine if the constraints can be satisfied under that labeling. Unfortunately, the type completion guess can be very expensive in practice.

Our presentation combines ideas from all of these algorithms as well as introducing some new ones. There is a set of upward and downward closure rules to mimic Oppen's algorithm. The idea of a type completion is replaced by a set of labeling rules that can be used to refine the set of possible constructors for each term (in particular, this allows us to delay guessing as long as possible). And the notion of introducing constructors and eliminating selectors is captured by a set of selector rules. In addition to the presentation, one of our key contributions is to provide precise side-conditions for when case splitting is necessary as opposed to when it can be delayed. The results given in Section 4 show that with the right strategy, significant gains in efficiency can be obtained.

We describe our procedure formally in the following, as a set of derivation rules. We build on and adopt the style of similar rules for abstract congruence closure [1] and syntactic unification [8].

3.1 Definitions and Notation

In the following, we will consider well-sorted formulas over the signature Σ above and an infinite set X of variables. To distinguish these variables, which can occur in formulas given to the decision procedure described below, from other internal variables used by the decision procedure, we will sometimes call the elements of X *input* variables.

Given a set Γ of literals over Σ and variables from X, we wish to determine the satisfiability of Γ in the algebra $\mathcal{R}$. We will assume for simplicity, and with no loss of generality, that the only occurrences of terms of sort bool are in atoms of the form $isC_k^j(t) \approx \mathsf{true}$, which we will write just as $isC_k^j(t)$.

Following [1], we will make use of the sets V_{τ_i} (V_{σ_i}) of *abstraction* variables of sort τ_i (σ_i); abstraction variables are disjoint from input variables (variables in Γ) and function as equivalence class representatives for the terms in Γ. We assume an arbitrary, but fixed, well-founded ordering $\succ$ on the abstraction variables that is total on variables of the same

sort. We denote the set of all variables (both input and abstraction) in E as $\mathcal{V}ar(E)$. We will use the expression $lbls(\tau_i)$ for the set $\{C^i_1, \ldots, C^i_{m_i}\}$ and define $lbls(\sigma_l)$ to be the empty set of labels for each σ_l. We will write $sort(t)$ to denote the sort of the term t.

The rules make use of three additional constructs that are not in the language of Σ: $\rightarrow$, $\mapsto$, and *Inst*.

The symbol $\rightarrow$ is used to represent *oriented* equations. Its left-hand side is a Σ-term t and its right-hand side is an abstraction variable v. The symbol $\mapsto$ denotes *labellings* of abstraction variables with sets of constructor symbols. It is used to keep track of possible constructors for instantiating a τ_i variable.[4] Finally, the *Inst* construct is used to track applications of the **Instantiate** rules given below. It is needed to ensure termination by preventing multiple applications of the same **Instantiate** rule. It is a unary predicate that is applied only to abstraction variables.

Let Σ^{C} denote the set of all constant symbols in Σ, including nullary constructors. We will denote by Λ the set of all possible literals over Σ and input variables X. Note that this does not include oriented equations ($t \rightarrow v$), labeling pairs ($v \mapsto L$), or applications of *Inst*. In contrast, we will denote by E multisets of literals of Λ, oriented equations, labeling pairs, and applications of *Inst*. To simplify the presentation, we will consistently use the following meta-variables: c, d denote constants (elements of Σ^{C}) or input variables from X; u, v, w denote abstraction variables; t denotes a *flat term*—i.e., a term all of whose proper sub-terms are abstraction variables—or a label set, depending on the context. $\mathbf{u}, \mathbf{v}$ denote possibly empty sequences of abstraction variables; and $\mathbf{u} \rightarrow \mathbf{v}$ is shorthand for the set of oriented equations resulting from pairing corresponding elements from $\mathbf{u}$ and $\mathbf{v}$ and orienting them so that the left hand variable is greater than the right hand variable according to $\succ$. Finally, $v \bowtie t$ denotes any of $v \approx t$, $t \approx v$, $v \not\approx t$, $t \not\approx v$, or $v \mapsto t$. To streamline the notation, we will sometimes denote function application simply by juxtaposition.

Each rule consists of a premise and one or more conclusions. Each premise is made up of a multiset of literals, oriented equations, labeling pairs, and applications of *Inst*. Conclusions are either similar multisets or $\bot$, where $\bot$ represents a trivially unsatisfiable formula. The soundness of our rule-based procedure depends on the fact that the premise E of a rule is satisfied in $\mathcal{R}$ by a valuation α of $\mathcal{V}ar(E)$ iff one of the conclusions E' of the rule is satisfied in $\mathcal{R}$ by an extension of α to $\mathcal{V}ar(E')$.

3.2 The derivation rules

Our decision procedure consists of the following derivation rules on multisets E.

Abstraction rules

Abstract 1
$$\frac{p[c],\ E}{c \rightarrow v,\ v \mapsto lbls(s),\ p[v],\ E} \quad \text{if} \quad \begin{array}{l} p \in \Lambda,\ c : s, \\ v \text{ fresh from } V_s \end{array}$$

Abstract 2
$$\frac{p[C^i_j\mathbf{u}],\ E}{C^i_j\mathbf{u} \rightarrow v,\ p[v],\ v \mapsto \{C^i_j\},\ E} \quad \text{if } p \in \Lambda,\ v \text{ fresh from } V_{\tau_i}$$

Abstract 3
$$\frac{p[S^i_{j,k}u],\ E}{\begin{array}{l} S^i_{j,1}u \rightarrow v_1,\ \ldots,\ S^i_{j,n^i_j}u \rightarrow v_{n^i_j},\ p[v_k], \\ v_1 \mapsto lbls(s_1),\ \ldots, v_{n^i_j} \mapsto lbls(s_{n^i_j}),\ E \end{array}} \quad \text{if} \quad \begin{array}{l} p \in \Lambda, S^i_{j,k} : \tau_i \rightarrow s_k, \\ \text{each } v_\iota \text{ fresh from } V_{s_\iota} \end{array}$$

[4] To simplify the writing of the rules, some rules may introduce labeling pairs for variables with a non-τ sort, even though these play no role.

The abstraction or *flattening* rules assign a new abstraction variable to every sub-term in the original set of literals. Abstraction variables are then used as place-holders or equivalence class representatives for those sub-terms. While we would not expect a practical implementation to actually introduce these variables, it greatly simplifies the presentation of the remaining rules. Notice that in each case, a labeling pair for the introduced variables is also created. This corresponds to labeling each sub-term with the set of possible constructors with which it could have been constructed. Also notice that in the **Abstract 3** rule, whenever a selector $S^i_{j,k}$ is applied, we effectively introduce all possible applications of selectors associated with the same constructor. This simplifies the later selector rules and corresponds to the step in the Oppen algorithm which ensures that in the term graph, any node with children has a complete set of children.

Literal level rules

Orient $\dfrac{u \approx v,\ E}{u \to v,\ E}$ if $u \succ v$ **Remove 1** $\dfrac{isC^i_j\, v,\ E}{v \mapsto \{C^i_j\},\ E}$

Inconsistent $\dfrac{v \not\approx v,\ E}{\bot}$ **Remove 2** $\dfrac{\neg isC^i_j\, v,\ E}{v \mapsto lbls(sort(v)) \setminus \{C^i_j\},\ E}$

The simple literal level rules are mostly self-explanatory. The **Orient** rule is used to replace an equation between abstraction variables (which every equation eventually becomes after applying the abstraction rules) with an oriented equation. Oriented equations are used in the remaining rules below. The **Remove** rules remove applications of testers and replace them with labeling pairs that impose the same constraints.

Upward (i.e., congruence) closure rules

Simplify 1 $\dfrac{u \bowtie t,\ u \to v,\ E}{v \bowtie t,\ u \to v,\ E}$ **Superpose** $\dfrac{t \to u,\ t \to v,\ E}{u \to v,\ t \to v,\ E}$ if $u \succ v$

Simplify 2 $\dfrac{f\mathbf{u}u\mathbf{v} \to w,\ u \to v,\ E}{f\mathbf{u}v\mathbf{v} \to w,\ u \to v,\ E}$ **Compose** $\dfrac{t \to v,\ v \to w,\ E}{t \to w,\ v \to w,\ E}$

These rules are modeled after similar rules for abstract congruence closure in [1]. The **Simplify** and **Compose** rules essentially provide a way to replace any abstraction variable with a smaller (according to $\succ$) one if the two are known to be equal. The **Superpose** rule merges two equivalence classes if they contain the same term. Congruence closure is achieved by these rules because if two terms are congruent, then after repeated applications of the first set of rules, they will become syntactically identical. Then the **Superpose** rule will merge their two equivalence classes.

Downward (i.e., unification) closure rules

Decompose $\dfrac{C^i_j\mathbf{u} \to v,\ C^i_j\mathbf{v} \to v,\ E}{C^i_j\mathbf{u} \to v,\ \mathbf{u} \to \mathbf{v},\ E}$

Clash $\dfrac{c \to v,\ d \to v,\ E}{\bot}$ if $c, d \in \Sigma^{\mathrm{C}},\ c : \sigma,\ d : \sigma,\ c \neq d$

Cycle $\dfrac{C^{i_n}_{j_n}\mathbf{u_n}u\mathbf{v_n} \to u_{n-1}, \ldots, C^{i_2}_{j_2}\mathbf{u_2}u_2\mathbf{v_2} \to u_1, C^{i_1}_{j_1}\mathbf{u_1}u_1\mathbf{v_1} \to u, E}{\bot}$ if $n \geq 1$

The main downward closure rule is the **Decompose** rule: whenever two terms with the same constructor are in the same equivalence class, their arguments must be equal. The **Clash** rule detects instances of terms that are in the same equivalence class that must be disequal in the intended model. The **Cycle** rule detects the (inconsistent) cases in which a term would have to be cyclical.

Selector rules

Instantiate 1 $\dfrac{S^i_{j,1}u \to u_1,\ \ldots,\ S^i_{j,n^i_j}u \to u_{n^i_j},\ u \mapsto \{C^i_j\},\ E}{C^i_j u_1 \cdots u_{n^i_j} \to u,\ u \mapsto \{C^i_j\},\ \mathit{Inst}(u),\ E}$ if $\mathit{Inst}(u) \notin E$

Instantiate 2 $\dfrac{v \mapsto \{C^i_j\},\ E}{\begin{array}{l} C^i_j u_1 \cdots u_{n^i_j} \to v,\ \mathit{Inst}(v),\ E \\ u_1 \mapsto \mathit{lbls}(s^i_{j,1}), \ldots, u_{n^i_j} \mapsto \mathit{lbls}(s^i_{j,n^i_j}) \end{array}}$ if $\begin{array}{l} \mathit{Inst}(v) \notin E, \\ v \mapsto L \notin E, \\ C^i_j \text{ finite constructor}, \\ S^a_{b,c}(v) \to v' \notin E, \\ u_k \text{ fresh from } V_{s^i_{j,k}} \end{array}$

Collapse 1 $\dfrac{C^i_j u_1 \cdots u_{n^i_j} \to u,\ S^i_{j,k}u \to v,\ E}{C^i_j u_1 \cdots u_{n^i_j} \to u,\ u_k \approx v,\ E}$

Collapse 2 $\dfrac{S^i_{j,k}u \to v,\ u \mapsto L,\ E}{t^i_{j,k} \approx v,\ u \mapsto L,\ E}$ if $C^i_j \notin L$

Rule **Instantiate 1** is used to eliminate selectors by replacing the argument of the selectors with a new term constructed using the appropriate constructor. Notice that only terms that have selectors applied to them can be instantiated and then only once they are unambiguously labeled. Rule **Instantiate 2** is used for finite constructors. For completeness, terms labeled with finite constructors must be instantiated even when no selectors are applied to them. The **Collapse** rules eliminate selectors when the result of their application can be determined. In **Collapse 1**, a selector is applied to a term known to be equal to a constructor of the "right" type. In this case, the selector expression is replaced by the appropriate argument of the constructor. In **Collapse 2**, a selector is applied to a term which must have been constructed with the "wrong" constructor. In this case, the designated term $t^i_{j,k}$ for the selector replaces the selector expression.

Labeling rules

Refine $\dfrac{v \mapsto L_1,\ v \mapsto L_2,\ E}{v \mapsto L_1 \cap L_2,\ E}$ **Empty** $\dfrac{v \mapsto \emptyset,\ E}{\bot}$ if $v : \tau_i$

Split 1 $\dfrac{S^i_{j,k}(u) \to v,\ u \mapsto \{C^i_j\} \cup L,\ E}{S^i_{j,k}(u) \to v,\ u \mapsto \{C^i_j\},\ E \qquad S^i_{j,k}(u) \to v,\ u \mapsto L,\ E}$ if $L \neq \emptyset$

Split 2 $\dfrac{u \mapsto \{C^i_j\} \cup L,\ E}{u \mapsto \{C^i_j\},\ E \qquad u \mapsto L,\ E}$ if $\begin{array}{l} L \neq \emptyset, \\ \{C^i_j\} \cup L \text{ all finite constructors} \end{array}$

The **Refine** rule simply combines labeling constraints that may arise from different sources for the same equivalence class. **Empty** enforces the constraint that every τ-term must be constructed by some constructor. The splitting rules are used to refine the set of possible constructors for a term and are the only rules that cause branching. If a term labeled with only finite constructors cannot be eliminated in some other way, **Split 2** must be applied until it is labeled unambiguously. For other terms, the **Split 1** rule only needs to be applied to distinguish the case of a selector being applied to the "right" constructor from a selector being applied to the "wrong" constructor. On either branch, one of the **Collapse** rules will apply immediately. We discuss this further in Section 4, below. The rules are proved sound, complete and terminating in our full report [4].

4 Strategies and Efficiency

It is not difficult to see that the problem of determining the satisfiability of an arbitrary set of literals is NP-complete. The problem was shown to be NP-hard in [12]. To see that it is in NP, we note that given a type completion, no additional splits are necessary, and the remaining rules can be carried out in polynomial time. However, as with other NP-complete problems (Boolean satisfiability being the most obvious example), the right strategy can make a significant difference in practical efficiency.

4.1 Strategies

A *strategy* is a predetermined methodology for applying the rules. Before discussing our recommended strategy, it is instructive to look at the closest related work. Oppen's original algorithm is roughly equivalent to the following: After abstraction, apply the selector rules to eliminate all instances of selector symbols. Next, apply upward and downward closure rules (the bidirectional closure). As you go, check for conflicts using the rules that can derive $\bot$. We will call this the *basic strategy*. Note that it excludes the splitting rules: because Oppen's algorithm assumes a single constructor, the splitting rules are never used. A generalization of Oppen's algorithm is mentioned in [14]. They add the step of initially guessing a "type completion". To model this, consider the following simple **Split** rule:

$$\textbf{Split} \quad \frac{u \mapsto \{C_j^i\} \cup L,\ E}{u \mapsto \{C_j^i\},\ E \qquad\qquad u \mapsto L,\ E} \quad \text{if} \quad L \neq \emptyset$$

Now consider a strategy which invokes **Split** greedily (after abstraction) until it no longer applies and then follows the basic strategy. We will call this strategy the *greedy splitting* strategy. One of the key contributions of this paper is to recognize that the greedy splitting strategy can be improved in two significant ways. First, the simple **Split** rule should be replaced with the smarter **Split 1** and **Split 2** rules. Second, these rules should be delayed as long as possible. We call this the *lazy splitting* strategy. The lazy strategy reduces the size of the resulting derivation in two ways. First, notice that **Split 1** is only enabled when some selector is applied to u. By itself, this eliminates many needless case splits. Second, by applying the splitting rules *lazily* (in particular by first applying selector rules), it may be possible to avoid splitting completely in many cases.

Example. Consider the following simple *tree* data type: $\mathit{tree} := \mathit{node}(\mathit{left} : \mathit{tree},\ \mathit{right} : \mathit{tree}) \mid \mathit{leaf}$ with *leaf* as the designated term for both selectors. Suppose we receive the input formula $\mathit{left}^n(Z) \approx X \ \wedge\ \mathit{is\_node}(Z) \ \wedge\ Z \approx X$. After applying all available rules except for the splitting rules, the result will look something like this:

Worst Case Number of Splits	Benchmarks	Sat	Unsat	Greedy		Lazy	
				Splits	Time (s)	Splits	Time (s)
0	4416	306	4110	0	24.6	0	24.6
1-5	2520	2216	304	6887	16.8	2414	17.0
6-10	692	571	121	4967	5.8	1597	5.7
11-20	178	112	66	2422	2.3	517	1.6
21-100	145	73	72	6326	4.5	334	1.1
101+	49	11	38	16593	9.8	73	0.3

Table 1: Greedy vs. Lazy Splitting

$$\begin{array}{ll}\{ & Z \to u_0, X \to u_0, u_0 \mapsto \{node\}, node(u_1, v_1) \to u_0, u_n \to u_0, \\ & left(u_1) \to u_2, \ldots, left(u_{n-1}) \to u_n, u_1 \mapsto \{leaf, node\}, \ldots, u_n \mapsto \{leaf, node\}, \\ & right(u_1) \to v_2, \ldots, right(u_{n-1}) \to v_n, v_1 \mapsto \{leaf, node\}, \ldots, v_n \mapsto \{leaf, node\}\},\end{array}$$

Notice that there are $2n$ abstraction variables labeled with two labels each. If we eagerly applied the naive **Split** rule at this point, the derivation tree would reach size $O(2^{2n})$.

Suppose, on the other hand, that we use the lazy strategy. First notice that **Split 1** can only be applied to n of the abstraction variables ($u_i, 1 \leq i \leq n$). Thus the more restrictive side-conditions of **Split 1** already reduce the size of the problem to at worst $O(2^n)$ instead of $O(2^{2n})$. However, by only applying it lazily, we do even better: suppose we split on u_i. The result is two branches, one with $u_i \mapsto \{node\}$ and the other with $u_i \mapsto \{leaf\}$. The second branch induces a cascade of (at most n) applications of Collapse 2 which in turn results in $u_k \mapsto \{leaf\}$ for each $k > i$. This eventually results in $\perp$ via the Empty and Refine rules. The other branch contains $u_i \mapsto \{node\}$ and results in the application of the Instantiate rule, but little else, and so we will have to split again, this time on a different u_i. This process will have to be repeated until we have split on all of the u_i. At that point, there will be a cycle from u_0 back to u_0, and so we will derive $\perp$ via the Cycle rule. Because each split only requires at most $O(n)$ rules and there are $n - 1$ splits, the total size of the derivation tree will be $O(n^2)$.

4.2 Experimental Results

We have implemented both the lazy and the greedy splitting strategies in the theorem prover CVC Lite [2]. Using the mutually recursive data types *nat*, *list*, and *tree* mentioned in the introduction, we randomly generated 8000 benchmarks.[5]

As might be expected with a large random set, most of the benchmarks are quite easy. In fact, over half of them are solved without any case splitting at all. However, a few of them did prove to be somewhat challenging (at least in terms of the number of splits required). Table 1 shows the total time and case splits required to solve the benchmarks. The benchmarks are divided into categories based on the the maximum number of case splits required to solve the benchmark.

For easy benchmarks that don't require many splits, the two algorithms perform almost identically. However, as the difficulty increases, the lazy strategy performs much better. For the hardest benchmarks, the lazy strategy outperforms the greedy strategy by more than an order of magnitude.

[5]See `http://www.cs.nyu.edu/~barrett/datatypes` for details on the benchmarks and results.

5 Extending the Algorithm

In this section we briefly discuss several ways in which our algorithm can be used as a component in solving a larger or related problem.

5.1 Finite Sorts

Here we consider how to lift the limitation imposed before that each of $\sigma \in \{\sigma_1, \ldots, \sigma_r\}$ is infinite valued. Since we have no such restrictions on sorts τ_i, the idea is to simply replace such a σ by a new τ-like sort τ_σ, whose set of constructors (all of which will be nullary) will match the domain of σ. For example, if σ is a finite scalar of the form $\{1, \ldots, n\}$, then we can let $\tau_\sigma ::== null_1 \mid \ldots \mid null_n$. We then proceed as before, after replacing all occurrences of σ by τ_σ and each i by $null_i$.

5.2 Simulating Partial Function Semantics

As mentioned earlier, it is not clear how best to interpret the application of a selector to the wrong constructor. One compelling approach is to interpret selectors as partial functions. An evaluation of a formula then has three possible outcomes: true, false, or undefined. This approach may be especially valuable in a verification application in which application of selectors is required to be guarded so that no formula should ever be undefined. This can easily be implemented by employing the techniques described in [6]: given a formula to check, a special additional formula called a type-correctness condition is computed (which can be done in time and space linear in the size of the input formula). These two formulas can then be checked using a decision procedure that interprets the partial functions (in this case, the selectors) in some arbitrary way over the undefined part of the domain. The result can then be interpreted to reveal whether the formula would have been true, false, or undefined under the partial function semantics.

5.3 Cooperating with other Decision Procedures

A final point is that that our procedure has been designed to integrate easily into a Nelson-Oppen-style framework for cooperating decision procedures [10]. In the many-sorted case, the key theoretical requirements (see [13]) for two decision procedures to be combined are that the signatures of their theories share at most sort symbols and each theory is *stably infinite* over the shared sorts.[6] A key operational requirement is that the decision procedure is also able to easily compute and communicate equality information.

The theory of $\mathcal{R}$ (i.e., the set of sentences true in $\mathcal{R}$) is trivially stably infinite over the sorts $\sigma_1, \ldots, \sigma_r$ and over any τ-sort containing a non-finite constructor—as all such sorts denote infinite sets in $\mathcal{R}$. Also, in our procedure the equality information is eventually completely captured by the oriented equations produced by the derivation rules, and so entailed equalities can be easily detected and reported.

For a detailed and formal discussion of how to integrate a rule-based decision procedure such as this one into a general framework combining Boolean reasoning and multiple decision procedures, we refer the reader to our related work in [3]. Note that, in particular, this work shows how the internal theory case splits can be delegated on demand to the Boolean engine; this is the implementation strategy followed in CVC Lite.

[6] A many-sorted theory T is stably infinite over a subset S of its sorts if every quantifier-free formula satisfiable in T is satisfiable in a model of T where the sorts of S denote infinite sets.

References

[1] L. Bachmair, A. Tiwari, and L. Vigneron. Abstract congruence closure. *JAR*, 31:129–168, 2003.

[2] C. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *Proceedings of CAV*, pages 515–518, July 2004.

[3] C. Barrett, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Splitting on demand in satisfiability modulo theories. Technical report, University of Iowa, 2006. Available at `ftp://ftp.cs.uiowa.edu/pub/tinelli/papers/BarNOT-RR-06.pdf`.

[4] C. Barrett, I. Shikanian, and C. Tinelli. An abstract decision procedure for satisfiability in the theory of recursive data types. Technical Report TR2005-878, Department of Computer Science, New York University, Nov. 2005.

[5] M. P. Bonacina and M. Echenim. Generic theorem proving for decision procedures. Technical report, Università degli studi di Verona, 2006. Available at `http://profs.sci.univr.it/~echenim/`.

[6] S. B. et al. A practical approach to partial functions in CVC Lite. In W. A. et al., editor, *Selected Papers from PDPAR '04*, volume 125(3) of *ENTCS*, pages 13–23. Elsevier, July 2005.

[7] W. Hodges. *A Shorter Model Theory*. Cambridge University Press, 1997.

[8] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.

[9] K. Meinke and J. V. Tucker. Universal algebra. In S. Abramsky, D. V. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 1. Claredon Press, 1992.

[10] G. Nelson and D. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–57, 1979.

[11] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *JACM*, 27(2):356–364, April 1980.

[12] D. C. Oppen. Reasoning about recursively defined data structures. *JACM*, 27(3):403–411, July 1980.

[13] C. Tinelli and C. Zarba. Combining decision procedures for sorted theories. In J. Alferes and J. Leite, editors, *Proceedings of JELIA '04*, volume 3229 of *LNAI*, pages 641–653. Springer, 2004.

[14] T. Zhang, H. B. Sipma, and Z. Manna. Decision procedures for term algebras with integer constraints. In *Proceedings of IJCAR '04 LNCS 3097*, pages 152–167, 2004.

[15] T. Zhang, H. B. Sipma, and Z. Manna. Term algebras with length function and bounded quantifier alternation. In *Proceedings of TPHOLs*, 2004.

A Framework for Decision Procedures in Program Verification

(presentation-only paper)

Ofer Strichman[1] and **Daniel Kroening**[2]
[1] Technion, Haifa, Israel
[2] ETH Zurich, Switzerland

ofers@ie.technion.ac.il, daniel.kroening@inf.ethz.ch

Abstract

Program analysis tools for modern programming languages require a very rich logic to reason about constructs such as bit-vector operators or dynamic data structures. We propose a generic framework for reducing decidable decision problems to propositional logic that starts with an axiomatization of the logics. Instantiating the framework for a specific logic requires a deductive decision procedure that fulfills several conditions. Linear arithmetic, the theory of arrays and other logics useful for verification, have such a decision procedure, as we show. Further, the framework allows to reduce combined theories to a unified propositional formula, which enables learning across theories. We conclude by presenting several challenges that need to be met for making the framework more general and useful for program verification.

Easy Parameterized Verificaton of Biphase Mark and 8N1 Protocols

(presentation-only paper)

Geoffrey Brown[1] **and Lee Pike** [2]

[1] Indiana University
[2] Galois Connections
geobrown@cs.indiana.edu, leepike@galois.com

Abstract

The Biphase Mark Protocol (BMP) and 8N1 Protocol are physical layer protocols for data transmission. We present a generic model in which timing and error values are parameterized by linear constraints, and then we use this model to verify these protocols. The verifications are carried out using SRI's SAL model checker that combines a satisfiability modulo theories decision procedure with a bounded model checker for highly-automated induction proofs of safety properties over infinite-state systems. Previously, parameterized formal verification of real-time systems required mechanical theorem-proving or specialized real-time model checkers; we describe a compelling case-study demonstrating a simpler and more general approach. The verification reveals a significant error in the parameter ranges for 8N1 given in a published application note.

Predicate Learning and Selective Theory Deduction for Solving Difference Logic

(presentation-only paper)

Chao Wang, Aarti Gupta, Malay Ganai
NEC Labs America, Princeton, USA
{chaowang|agupta|malay}@nec-labs.com

Abstract

Design and verification of systems at the Register-Transfer (RT) or behavioral level require the ability to reason at higher levels of abstraction. Difference logic consists of an arbitrary Boolean combination of propositional variables and difference predicates and therefore provides an appropriate abstraction. In this paper, we present several new optimization techniques for efficiently deciding difference logic formulas. We use the lazy approach by combining a DPLL Boolean SAT procedure with a dedicated graph-based theory solver, which adds transitivity constraints among difference predicates on a "need-to" basis. Our new optimization techniques include flexible theory constraint propagation, selective theory deduction, and dynamic predicate learning. We have implemented these techniques in our lazy solver. We demonstrate the effectiveness of the proposed techniques on public benchmarks through a set of controlled experiments.

Deciding Extensions of the Theory of Arrays by Integrating Decision Procedures and Instantiation Strategies

(presentation-only paper)

Silvio Ghilardi[1], **Enrica Nicolini**[1],
Silvio Ranise[2] **and Daniele Zucchelli**[1]
[1] Università degli Studi di Milano, Italy
[2] LORIA and INRIA-Lorraine, Nancy, France
ghilardi@dsi.unimi.it, nicolini@mat.unimi.it,
silvio.ranise@loria.fr, zucchelli@dsi.unimi.it

Abstract

The theory of arrays, introduced by McCarthy in his seminal paper "Toward a mathematical science of computation", is central to Computer Science. Unfortunately, the theory alone is not sufficient for many important verification applications such as program analysis. Motivated by this observation, we study extensions of the theory of arrays whose satisfiability problem (i.e. checking the satisfiability of conjunctions of ground literals) is decidable. In particular, we consider extensions where the indexes of arrays has the algebraic structure of Presburger Arithmetic and the theory of arrays is augmented with axioms characterizing additional symbols such as dimension, sortedness, or the domain of definition of arrays.

We provide methods for integrating available decision procedures for the theory of arrays and Presburger Arithmetic with automatic instantiation strategies which allow us to reduce the satisfiability problem for the extension of the theory of arrays to that of the theories decided by the available procedures. Our approach aims to reuse as much as possible existing techniques so to ease the implementation of the proposed methods. To this end, we show how to use both model-theoretic and rewriting-based theorem proving (i.e., superposition) techniques to implement the instantiation strategies of the various extensions.

Producing Conflict Sets for Combinations of Theories

(presentation-only paper)

Silvio Ranise, **Christophe Ringeissen** and **Duc-Khanh Tran**
LORIA and INRIA-Lorraine, Nancy, France
{ranise|ringeiss|tran}@loria.fr

Abstract

Recently, it has become evident that the capability of computing conflict sets is of paramount importance for the efficiency of systems for Satisfiability Modulo Theories (SMT). In this paper, we consider the problem of modularly constructing conflict sets for a combined theory, when this is obtained as the disjoint union of many component theories for which satisfiability procedures capable of computing conflict sets are assumed available. The key idea of our solution to this problem is the concept of explanation graph, which is a labelled, acyclic and undirected graph capable of recording the entailment of some elementary equalities. Explanation graphs allow us to record the explanations computed by, for instance, a proof-producing congruence closure and to extend the well-known Nelson-Oppen combination method to modularly build conflict sets for disjoint unions of theories. We study how the computed conflict sets relate to an appropriate notion of minimality.