The Strategy Challenge in SMT Solving

Leonardo de Moura[1] and Grant Olney Passmore[2,3]
leonardo@microsoft.com, grant.passmore@cl.cam.ac.uk

[1] Microsoft Research, Redmond
[2] Clare Hall, University of Cambridge
[3] LFCS, University of Edinburgh

Abstract. High-performance SMT solvers contain many tightly integrated, hand-crafted heuristic combinations of algorithmic proof methods. While these heuristic combinations tend to be highly tuned for known classes of problems, they may easily perform badly on classes of problems not anticipated by solver developers. This issue is becoming increasingly pressing as SMT solvers begin to gain the attention of practitioners in diverse areas of science and engineering. We present a challenge to the SMT community: to develop methods through which users can exert strategic control over core heuristic aspects of SMT solvers. We present evidence that the adaptation of ideas of strategy prevalent both within the Argonne and LCF theorem proving paradigms can go a long way towards realizing this goal.

Prologue. Bill McCune, Kindness and Strategy, by Grant Passmore

I would like to tell a short story about Bill, of how I met him, and one way his work and kindness impacted my life.

I was an undergraduate at the University of Texas at Austin, where in Autumn 2004 I was lucky enough to take an automated reasoning course with Bob Boyer. One of our three main texts for the course (the other two being Robinson's 1965 JACM article on resolution and part of Gödel's 1931 Incompleteness paper in which he defines his primitive recursive proof checker) was the wonderful book[4] of Larry Wos and Gail Pieper on Bill's OTTER theorem prover. I was mesmerized by OTTER's power and flexibility, and seduced by the playful way their book taught us to apply it to real problems.

At the start of the Summer that followed, I began an internship at a company, National Instruments in Austin, centered around the application of theorem provers to LabVIEW/G, a concurrent, graphical programming language. Given my exposure to them, OTTER and the Boyer-Moore prover ACL2 were the first tools I applied to the job. Soon after I began, I contacted Bill and asked his advice. He not only gave me help and direction, but invited me to come to Argonne National Laboratory and give a talk. The first talk I ever gave on my own research was at Argonne at Bill's invitation.

[4] *A Fascinating Country in the World of Computing: Your Guide to Automated Reasoning* by Larry Wos and Gail W. Pieper, World Scientific, 2000.

I can still remember the excitement I felt at flying into Chicago, knowing that I would soon get to meet and brainstorm with Bill and Larry, true heroes of mine. At some point during my visit, Bill took notice of how interested I was in one of Larry and Gail's new monographs. It was a book[5] on *proof strategy*, the study of how one might navigate (as Dolph Ulrich writes in his beautiful Foreword) the "unimaginably vast" sea of deductions, "a space in which all proofs solving a particular problem at hand might well be as unreachable as the farthest stars in the most distant galaxies." Bill also must have noticed that the price of the book was very steep for me as an undergraduate, that it would take me some time to save up to afford a copy. A few weeks later, imagine my surprise when I received a package in the mail consisting of the book, a present from Bill. I have it with me in my office to this day. I am certain his kindness played a role in my going into automated reasoning.

The work that Leo and I present in this paper has been hugely influenced by the work of Bill, Larry and the Argonne group. Their relentless championing of the importance of strategy in resolution theorem proving has made the solution of so many otherwise impossible problems a reality. Our dream with this paper is to translate their important lesson and conviction into a form suitable for a comparatively young branch of automated reasoning known as SMT.

1 Introduction

SMT (Satisfiability Modulo Theories) solvers are a powerful class of automated theorem provers which have in recent years seen much academic and industrial uptake [19]. They draw on some of the most fundamental discoveries of computer science and symbolic logic. They combine the Boolean Satisfiability problem with the decision problem for concrete domains such as arithmetical theories of the integers, rationals and reals, and theories of data structures fundamental in computing such as finite lists, arrays, and bit-vectors. Research in SMT involves, in an essential way, decision problems, completeness and incompleteness of logical theories and complexity theory.

The standard account of modern SMT solver architecture is given by the so-called *DPLL(T) scheme* [35]. DPLL(T) is a theoretical framework, a rule-based formalism describing, abstractly, how satellite *theory solvers* (T-solvers) for decidable theories such as linear integer arithmetic, arrays and bit-vectors are to be integrated together with DPLL-based SAT solving. Decision procedures (complete T-solvers) for individual theories are combined by the DPLL(T) scheme in such a way that guarantees their combination is a complete decision procedure as well. Because of this, one might get the impression that *heuristics* are not involved in SMT. However, this is not so: heuristics play a vital role in high-performance SMT, a role which is all too rarely discussed or championed.

By design, DPLL(T) abstracts away from many practical implementation issues. High-performance SMT solvers contain many tightly integrated, hand-

[5] *Automated Reasoning and the Discovery of Missing and Elegant Proofs* by Larry Wos and Gail W. Pieper, Rinton Press, 2003

crafted heuristic combinations of algorithmic proof methods which fall outside the scope of DPLL(T). We shall discuss many examples of such heuristics in this paper, with a focus on our tools **RAHD** [36] and **Z3** [18]. To mention but one class of examples, consider formula preprocessing. This is a vital, heavily heuristic component of modern SMT proof search which occurs outside the purview of DPLL(T). We know of no two SMT solvers which handle formula preprocessing in exactly the same manner. We also know of no tool whose documentation fully describes the heuristics it uses in formula preprocessing, let alone gives end-users principled methods to control these heuristics.

While the heuristic components of SMT solvers tend to be highly tuned for known classes of problems (e.g., SMT-LIB [4] benchmarks), they may easily perform very badly on new classes of problems not anticipated by solver developers. This issue is becoming increasingly pressing as SMT solvers begin to gain the attention of practitioners in diverse areas of science and engineering. In many cases, changes to the prover heuristics can make a tremendous difference in the success of SMT solving within new problem domains. Classically, however, much of the control of these heuristics has been outside the reach[6] of solver end-users[7]. We would like much more control to be placed in the hands of end-users, and for this to be done in a principled way.

We present a challenge to the SMT community:

The Strategy Challenge

> To build theoretical and practical tools allowing users to exert strategic control over core heuristic aspects of high-performance SMT solvers.

In this way, high-performance SMT solvers may be tailored to specific problem domains, especially ones very different from those normally considered. We present evidence, through research undertaken with the tools **RAHD** and **Z3**, that the adaptation of a few basic ideas of strategy prevalent both within the Argonne and LCF theorem proving paradigms can go a long way towards realizing this goal. In the process, we solidify some foundations for strategy in the context of SMT and pose a number of questions and open problems.

[6] Some SMT solvers such as **CVC3** [5], **MathSAT** [10] and **Z3** [18] expose a vast collection of parameters to control certain behavioral aspects of their core proof procedures. We view these parameters as a primitive way of exerting strategic control over the heuristic aspects of high-performance SMT solvers. As the use of SMT solvers continues to grow and diversify, the number of these options has steadily risen in most solvers. For instance, the number of such options in **Z3** has risen from 40 (v1.0) to 240 (v2.0) to 284 (v3.0). Many of these options have been requested by end-users. Among end-users, there seems to be a wide-spread wish for more methods to exert strategic control over the prover's reasoning.

[7] We use the term *end-user* to mean a user of an SMT solver who does not contribute to the essential development of such a solver. End-users regularly embed SMT solvers into their own tools, making SMT solvers a subsidiary theorem proving engine for larger, specialised verification tool-chains.

1.1 Caveat Emptor: What This Paper Is and Is Not

We find it prudent at the outset to make clear what this paper is and is not. Let us first enumerate a few things we shall not do:

- We shall not give a new *theoretical framework* or *rule-based formalism* capturing the semantics of heuristic proof strategies, as is done, for instance, in the influential STRATEGO work on term rewriting strategies [30].
- We shall not prove any theorems about the algebraic structure underlying the collection of heuristic proof strategies, as is done, for instance, in the insightful work on the *proof monad* for interactive proof assistants [27].
- We shall not propose a concrete syntax for heuristic proof strategies in the context of SMT, as, for instance, an extension of the SMT-LIB standard [4].

We shall not attempt any of the (worthy) goals mentioned above because we believe to do so would be premature. It is simply too early to accomplish them in a compelling way. For example, before a standard concrete syntax is proposed for heuristic SMT proof strategies, many different instances of strategy in SMT must be explicated and analyzed, in many different tools, from many contrasting points of view. Before any theorems are proved about the collection of heuristic SMT proof strategies, we must have a firm grasp of their scope and limitations. And so on. Instead, our goals with this Strategy Challenge are much more modest:

- To bring awareness to the crucial role heuristics play in high-performance SMT, and to encourage researchers in the field to be more explicit as to the heuristic strategies involved in their solvers.
- To convince SMT solver developers that providing flexible methods (i.e., a *strategy language*) for end-users to exert fine-grained control over heuristic aspects of their solvers is an important, timely and worthy undertaking.
- To show how the adaptation of some ideas of strategy prevalent both within the Argonne and LCF theorem proving paradigms can go a long way towards realizing these goals.
- To stress that from a scientific point of view, the explication of the actual heuristic strategies used in high-performance SMT solvers is absolutely crucial for enabling the reproducibility of results presented in publications. For instance, if a paper reports on a new decision procedure, including experimental findings, and these experiments rely upon an implementation of the described decision method incorporating some heuristic strategies, then these heuristics should be communicated as well.

Throughout this paper, we shall present many examples in (variants of) the concrete syntaxes we have developed for expressing heuristic proof strategies in our own tools. These are given to lend credence to the practical value of this challenge, not as a proposal for a standard strategy language.

§

As we shall see, our initial approach for meeting this Strategy Challenge is based on an SMT-specific adaptation of the ideas of *tactics* and *tacticals* as found in LCF-style [33,39] proof assistants. This adaptation has a strong relationship with the approach taken by the Argonne group on *theorem proving toolkits*, work that began with the Argonne systems NIUTP1 - NIUTP7 in the 1970s and early 1980s, and has continued through to modern day with Bill McCune's OPS (OTTER Parts Store) in the early 1990s, the foundation of EQP and its parallel version Peers [31,9], and LADR (Library for Automated Deduction Research), the foundation of McCune's powerful Prover9 and MACE4 tools [32]. The way in which we adapt ideas of tactics and tacticals to SMT results in notions of strategy which, though borrowing heavily from both of these sources, are quite distinct from those found in LCF-style proof assistants and Argonne-style theorem provers.

Finally, let us assure the reader that we are *not* proposing a tactic-based approach for implementing "low-level" aspects of SMT solvers such as unit propagation, nor for implementing core reasoning engines such as a SAT solver or Simplex. This would hardly be practical for high-performance SMT. Again, our goals are more modest: We are proposing only the use of this strategy machinery for facilitating the orchestration of "big" reasoning engines, that is, for prescribing heuristic combinations of procedures such as Gaussian Elimination, Gröbner bases, CNF encoders, SAT solvers, Fourier-Motzkin quantifer elimination and the like. In this way, "big" symbolic reasoning steps will be represented as tactics heuristically composable using a language of tacticals.

While we do not propose a particular concrete strategy language for SMT solvers, we will present some key features we believe a reasonable SMT strategy language should have. These features have been implemented in our tools, and examples are presented. One important requirement will be that a strategy language support methods for conditionally invoking different reasoning engines based upon features of the formula being analyzed. We shall exploit this ability heavily. In some cases, when high-level "big" engines contain within themselves heavily heuristic components made up of combinations of other "big" reasoning engines, we also propose that these components be modularized and made replaceable by user-specified heuristic proof strategies given as parameters to the high-level reasoning engine. These ideas will be explained in detail in Sec. 3, where we give the basis of what we call *big-step strategies* in SMT, and in Sec. 4, where we observe that certain strategically parameterized reasoning engines can be realized as tacticals.

1.2 Overview

In Sec. 2 we begin with the question *What is strategy?* and explore some possible answers by recalling important notions of strategy in the history of automated theorem proving. In Sec. 3, we give the foundations of big-step strategies in SMT. In Sec. 4, we observe that there is a natural view of some reasoning engines as tacticals, and we present a few examples of this in the context of **RAHD** and **Z3**. In Sec. 5, we show some promising results of implementations of many of these strategy ideas within the two tools. Finally, we conclude in Sec. 6 with a look towards the future.

2 Strategy in Mechanized Proof

There exists a rich history of ideas of *strategy* in the context of mechanized proof. In this section, we work to give a high-level, incomplete survey of the many roles strategy has played in automated proof. In the process, we shall keep an eye towards why many of the same issues which motivated members of the non-SMT-based automated reasoning community to develop powerful methods for user-controllable proof search strategies also apply, compellingly, to the case of SMT.

2.1 What is Strategy?

Before discussing strategy any further, we should certainly attempt to define it. *What is strategy, after all?* Even restricted to the realm of mechanized proof, this question is terribly difficult to answer. There are so many aspects of strategy pervasive in modern proof search methods, and there seems to be no obvious precise delineations of their boundaries. Where does one, for instance, draw the line between the "strategic enhancement" of an existing search algorithm and the advent of a new search algorithm altogether?

Despite the difficulties fundamental to defining precisely what *strategy* is, many researchers in automated reasoning have proposed various approaches to incorporating strategy into automated proof. Some approaches have been quite foundational in character, shedding light on the nature of strategy in particular contexts. For instance, in the context of term rewriting, the ideas found within the STRATEGO system have given rise to much fruit, both theoretical and applied, and elucidated deep connections between term rewriting strategy and concurrency theory [30]. In the context of proof strategies in interactive theorem provers, the recent work of Kirchner-Muñoz has given heterogeneous proof strategies found in proof assistants like PVS a firm algebraic foundation using the category-theoretic notion of a monad [27].

Though a general definition of *what strategy is* seems beyond our present faculties[8], we may still make some progress by describing a few of its most salient aspects. In particular, the following two statements seem a reasonable place to begin:

1. There is a natural view of automated theorem proving as being an exercise in combinatorial search.
2. With this view in mind, then *strategy* may be something like *adaptations of general search mechanisms which reduce the search space by tailoring its exploration to a particular class of problems.*

[8] Much deep work outside the scope of this paper has been done on notions of strategy in automated reasoning, especially in the context of first-order theorem proving. See, e.g., the articles [7,8,28,41].

We are most interested in these adaptations when end-users of theorem proving tools may be given methods to control them.

To gain a more concrete understanding of the importance of strategy in automated theorem proving, it is helpful to consider some key examples of its use. In working to gather and present some of these examples, we have found the vast number of compelling uses of strategy to be quite staggering. There are so many examples, in fact, that it seems fair to say that much of the history of automated reasoning can be interpreted as a series of *strategic advances*[9]. As automated reasoning is such a broad field, let us begin by focusing on the use of strategy in one of its particularly telling strands, the history of mechanized proof search in first-order predicate calculus (FOL).

§

When the field of first-order proof search began and core search algorithms were first being isolated, most interesting strategic advancements were so profound that we today consider them to be the advent of genuinely "new" theorem proving methods. For example, both the Davis-Putnam procedure and Robinson's resolution can be seen to be strategic advancements for the general problem of first-order proof search based on Herbrand's Theorem. But, compared to their predecessors, the changes these strategic enhancements brought forth were of such a revolutionary nature that we consider them to be of quite a different kind than the strategies we want to make user-controllable in the context of SMT.

Once resolution became a dominant focus of first-order theorem proving, however, then more nuanced notions of strategy began to take hold, with each of them using resolution as their foundation. Many of the most lasting ideas in this line of research were developed by the Argonne group. These ideas, including the set of support, term weighting, the given-clause algorithm (and, e.g., its use of pick-given ratio), hot lists and hints, did something very interesting: They provided a fixed, general algorithmic search framework upon which end-users could exert some of their own strategic control by prescribing restrictions to guide the general method. Moreover, beginning in the 1970s, the Argonne group promoted the idea of *theorem proving toolkits*, libraries of high-performance reasoning engines one could use to build customized theorem provers. This idea has influenced very much the approach we propose for strategy in SMT.

Let us now turn, in more detail, to uses of strategy in the history of mechanized first-order proof. Following this, we shall then consider some aspects of

[9] As a very helpful reviewer of this paper pointed out to us, it is interesting to note that the boundary between "strategy" and "inference system" is a flexible one which has been constantly evolving in theorem proving. Take for instance ordered resolution: it is presently considered an inference system, but at its inception the notion of selecting literals based on an ordering could have been seen as a strategy. There are many examples of ideas which were born as "strategy" and then became full-fledged "inference systems" as they were formalized.

strategy in the context of LCF-style interactive proof assistants. Ideas of strategy from both of these histories will play into our work.

2.2 Strategy in Automated Theorem Proving over FOL

One cannot attempt to interpret the history of mechanized proof in first-order logic without bearing in mind the following fact: Over two decades before the first computer experiments with first-order proof search were ever performed, the undecidability of FOL was established. This result was well-known to the logicians who began our field. Thankfully, this seemingly negative result was tempered with a more optimistic one: the fact that FOL is *semi-decidable*. This allowed programs such as the British Museum Algorithm (and quickly others of a more sophisticated nature) to be imagined which, in principle, will always find a proof of a conjecture C over an axiomatic theory T if C is in fact true in all models of T.

Early in the field, and long before the advent of algorithmic complexity theory in any modern sense, obviously infeasible approaches like the British Museum [34] were recognized as such. Speaking of the earliest (1950s) research in computer mechanized proof, Davis writes in "The Early History of Automated Deduction" [14]:

> [...] it was all too obvious that an attempt to generate a proof of something non-trivial by beginning with the axioms of some logical system and systematically applying the rules of inference in all possible directions was sure to lead to a gigantic combinatorial explosion.

Thus, though a semi-complete theorem proving method was known (and such a method is in a sense "best possible" from the perspective of computability theory), its search strategy was seen as utterly hopeless. In its place, other search strategies were sought in order to make the theorem proving effort more tractable. This point of view was articulated at least as early as 1958 by Hao Wang, who writes in [45]:

> Even though one could illustrate how much more effective partial strategies can be if we had only a very dreadful general algorithm, it would appear desirable to postpone such considerations till we encounter a more realistic case where there is no general algorithm or no efficient general algorithm, e.g., in the whole predicate calculus or in number theory. As the interest is presumably in seeing how well a particular procedure can enable us to prove theorems on a machine, it would seem preferable to spend more effort on choosing the more efficient methods rather than on enunciating more or less familiar generalities.

At the famous 1957 five week Summer Institute for Symbolic Logic held at Cornell University, the logician Abraham Robinson[10] gave a remarkably influen-

[10] It is useful to note that this Abraham Robinson, the model theorist and inventor of non-standard analysis, is not the same person as John Alan Robinson who would some 8 years later invent the proof search method of first-order resolution.

tial short talk [43] in which he singled out Skolem functions and Herbrand's Theorem as potentially useful tools for general-purpose first-order theorem provers [14]. Aspects of this suggestion were taken up by many very quickly, notably Gilmore [22], Davis and Putnam [16], and eventually J.A. Robinson [44]. Let us examine, from the perspective of strategy, a few of the main developments in this exceedingly influential strand.

As noted by Davis [14], the first Herbrand-based theorem provers for FOL employed completely *unguided* search of the Herbrand universe. There was initially neither a top-level conversion to a normal form such as CNF nor a systematic use of Skolem functions. When these first Herbrand-based methods were applied, through for instance the important early implementation by Gilmore [22], they proved unsuccessful for all but the simplest of theorems. Contemporaneously, Prawitz observed the same phenomena through his early work on a prover based on a modified semantic tableaux [14]. The lesson was clear: unguided search, even when based on deep results in mathematical logic, is not a viable approach. Again, new *strategies* were sought for controlling search space exploration.

The flurry of theorem proving breakthroughs in the early 1960s led to a wealth of new search strategies (and new notions of strategy) which still form the foundation for much of our field today.

First, based on shortcomings they observed in Gilmore's unguided exploration of the Herband universe, in particular the reliance of his method upon a DNF conversion for (what we now call) SAT solving, Davis and Putnam devised a new Herbrand universe exploration strategy which systematically applied Skolemization to eliminate existential quantifiers and used a CNF input formula representation as the basis to introduce a number of enhanced techniques for recognising the unsatisfiability of ground instances [16]. In the process, they spent much effort on enhancing the tractable recognition of ground unsatisfiability, which they believed at the time to be the biggest practical hurdle in Herbrand-based methods [14]. When further enhanced by Davis, Logemann and Loveland, this propositional component of Davis and Putnam's general first-order search strategy gave us what we now call DPLL, the foundation of modern SAT solving [15]. Nevertheless, once implementations were undertaken and experiments performed, the power of their first-order method was still found completely unsatisfactory. As Davis states [14],

> Although testing for satisfiability was performed very efficiently, it soon became clear that no very interesting results could be obtained without first devising a method for avoiding the generation of spurious elements of the Herbrand universe.

In the same year, Prawitz devised an "on-demand" method by which the generation of unnecessary terms in Herbrand expansions could be avoided, at the cost of sequences of expensive DNF conversions [42]. Though these DNF conversions precluded the practicality of Prawitz's method, his new approach made clear the potential utility of unification in Herbrand-based proof search, and Davis soon proposed [13,14] that effort be put towards

> ... a new kind of procedure which seeks to combine the virtues of the Prawitz procedure and those of the Davis Putnam procedure.

Two years later, Robinson published his discovery of such a method: Resolution, a single, easily mechanizable inference rule (refutationally) complete for FOL. This new method soon became a dominant high-level strategy for first-order proof search [44]. That resolution was a revolutionary improvement over previous methods is without question. But, practitioners soon discovered that the game was by no means won. Even with resolution in hand, the act of proof search was utterly infeasible for the vast majority of nontrivial problems without the introduction of some techniques for guiding the generation of resolvents. It is here that a new class of strategies was born.

In the 1960s, the group centered around Larry Wos at Argonne National Laboratory contributed many fundamental developments to first-order proving. At Argonne, Robinson made his discovery of resolution, first communicated in a privately circulated technical report in 1964, and then published in his influential JACM paper the year that followed. Almost immediately, Wos and his colleagues championed the importance of user-controllable strategies during resolution proof search and began developing methods for their facilitation. At least two important papers in this line were published by the end of 1965: “The Unit Preference Strategy in Theorem Proving” [47] and “Efficiency and Completeness of the Set of Support Strategy in Theorem Proving” [48]. The first gave rise to a strategy that the resolution prover would execute without any influence from the user. The second, however, introduced a strategy of a different kind: a method by which end-users could exert strategic control over the proof search without changing its underlying high-level method or impinging upon its completeness.

In resolution, the idea of set of support is to partition the CNF representation of the negated input formula into two sets of clauses, a satisfiable set A and another set S, and then to restrict the generation of clauses to those who can trace their lineage back to at least one member of S. Importantly, the choice of this division of the input clauses between A and S may be chosen strategically by the end-user. This general approach to heuristic integration — parameterizing a fixed proof search method by user-definable data — has proven enormously influential. Other methods in this class include clause weighting, hot lists, pick-given ratio, and many others.

Beginning in the 1970s, the Argonne group made an important methodological decision. This was the introduction of *theorem proving toolkits*. As Lusk describes [29],

> The notion of a toolkit with which one could build theorem provers was introduced at this time and became another theme for later Argonne development. In this case the approach was used to build a series of systems incorporating ever more complex variations on the closure algorithm without changing the underlying data structures and inference functions. The ultimate system (NIUTP7) provided a set of user-definable theorem-proving “environments,” each running a version of the closure algorithm

> with different controlling parameters, and a meta-language for controlling their interactions. There was enough control, and there were enough built-in predicates, that it became possible to “program” the theorem prover to perform a number of symbolic computation tasks. With these systems, Winker and Wos began the systematic attack on open problems [...].

Finally, we shall end our discussion of strategy in first-order provers with a few high-level observations. Given that the whole endeavor is undecidable, researchers in first-order theorem proving recognized very early that strategy must play an indispensible role in actually finding proofs. In the beginning of the field, new strategies were often so different from their predecessors that we consider them to be genuinely new methods of proof search altogether. But, once a general method such as resolution had taken hold as a dominant foundation, then strategies were sought for allowing users to control specific aspects of this fixed foundation. Abstractly, this was accomplished by providing a resolution proof search loop which accepts strategic data to be given as user-specifiable parameters.

Let us turn our attention now to another important contributor to ideas of strategy in computer-assisted proof, the LCF-style of interactive proof assistants.

2.3 Strategy in LCF-style Interactive Proof Assistants

In the field of interactive proof assistants, strategy appears in many forms. The fact that humans contribute much more to the proof development process in proof assistants than in fully automatic provers gives rise to ideas of strategy quite distinct from those encountered in our discussion of first-order provers. As above, we must limit our discussion to a very small segment of the field. Let us in this section discuss one key exemplar of strategy in proof assistants, the approach of LCF.

Strategy in the LCF Approach The original LCF was an interactive proof checking system designed by Robin Milner at Stanford in 1972. This system, so-named for its mechanization of Scott’s Logic of Computable Functions, provided a proof checker with the following high-level functionality [23]:

> Proofs [were] conducted by declaring a main goal (a formula in Scott’s logic) and then splitting it into subgoals using a fixed set of subgoaling commands (such as induction to generate the basis and step). Subgoals [were] either solved using a simplifier or split into simpler subgoals until they could be solved directly.

Soon after the birth of Stanford LCF, Milner moved to Edinburgh and built a team to work on its successor. A number of shortcomings had been observed in the original system. In particular, Stanford LCF embodied only one high-level proof strategy: ‘backwards’ proof, working from goals to subgoals. Moreover, even within a backwards proof, Stanford LCF had only a fixed set of proof construction commands which could not be easily extended [23]. Finding principled techniques to free the system from these strategic shackles became a driving motivation behind the design of Edinburgh LCF.

To address these problems, Milner devised a number of ingenious solutions which still today form the design foundation for many widely-used proof assistants. Fundamentally, Edinburgh LCF took the point of view of treating proofs as computation. New theorems were to be computed from previously established theorems by a fixed set of theorem constructors. To ensure that this computation was always done in a correct way, Milner designed an abstract datatype `thm` whose predefined values were instances of axioms and whose constructors were inference rules [23]. The idea was that strict type-checking would guarantee soundness by making sure that values of type `thm` were always actual theorems. The strictly-typed programming language ML (the **M**eta **L**anguage of Edinburgh LCF) was then designed to facilitate programming *strategies* for constructing values of type `thm`. It was a remarkable achievement. The strategic shackles of Stanford LCF had most certainly been relinquished, but much difficult work remained in order to make this low-level approach to proof construction practical. Many core strategies, both for facilitating high-level proof methods like backwards proof, as well as for implementing proof methods such as simplifiers and decision procedures needed to be built before it would be generally useful to end-users.

As mentioned, with the bare foundation of a type `thm` and the meta language ML, the system did not directly support backwards proof. To remedy this, Milner introduced *tactics* and *tacticals*. The idea of backwards proof is that one begins with a goal, reduces it to simpler subgoals, and in the process forms a proof tree. When any of these subgoals have been made simple enough to be discharged, then a branch in the proof tree can be closed. A tactic reduces a goal to a set of subgoals such that if every subgoal holds then the goal also holds. If the set of unproved subgoals is empty, then the tactic has proved the goal. A tactic not only reduces a goal to subgoals, but it also returns a proof construction function to justify its action. *Tacticals* are combinators that treat tactics as data, and are used to construct more complex tactics from simpler ones. Gordon summarizes nicely [23]:

> By encoding the logic as an abstract type, Edinburgh LCF directly supported forward proof. The design goal was to implement goal directed proof tools by programs in ML. To make ML convenient for this, the language was made functional so that subgoaling strategies could be represented as functions (called "tactics" by Milner) and operations for combining strategies could be programmed as higher-order functions taking strategies as arguments and returning them as results (called "tacticals"). It was anticipated that strategies might fail (e.g. by being applied to inappropriate goals) so an exception handling mechanism was included in ML.

Since the time of Edinburgh LCF (and its soon-developed successor, Cambridge LCF), the technology within LCF-style proof assistant has grown considerably. However, the underlying design principles centered around user-definable proof strategies have remained more-or-less the same, and are found today in tools like Isabelle [40], HOL [24], Coq [6], MetaPRL [25] and Matita [3]. For each of these tools, immense work has gone into developing powerful tactics which embody particular proof strategies. Many of them, such as the proof producing real closed field quantifier elimination tactic in HOL-Light, are tactics embodying complete decision procedures for certain logical theories. Others, such as the implementation by Boulton of a tactic based on Boyer-Moore induction heuristics in HOL, are powerful, incomplete heuristic strategies working over undecidable theories.

There are many things to learn from the success of the LCF paradigm. One lesson relevant to our work is the following: By "opening up" strategic aspects of the proof effort and providing principled, sound programming methods through which users may write their own proof strategies, it has been possible for enormously diverse ecosystems of powerful proof strategies to be developed, contributed and shared by communities of users of LCF-style proof assistants. As these ecosystems grow, the theorem proving tools become stronger, and the realizable practical verification efforts scale up significantly more than they would if these user-specifiable strategic enhancements were not possible.

3 Foundations for Big-Step Strategies

In this section, we propose a methodology for orchestrating reasoning engines where "big" symbolic reasoning steps are represented as functions known as *tactics*, and tactics are composed using combinators known as *tacticals*. We define notions of goals, tactics and tacticals in the context of SMT. Our definitions diverge from the ones found in LCF for the following main reasons:

- in SMT, we are not only interested in proofs of goals, but also in counter-examples (models yielding satisfying instances), and
- we want to support over and under-approximations when defining strategies.

Goals. The SMT decision problem consists of deciding whether a set of formulas S is satifisfiable or not modulo some background theory. We say each one of the formulas in S is an *assumption.* This observation suggests that a goal might be simply a collection of formulas. For practical reasons, a goal will also have a collection of attributes. Later, we describe some of the attributes we use in our systems. Thus, a goal is a pair comprised of a sequence of formulas and a sequence of attributes. We use sequences instead of sets because the ordering is relevant for some operations in our framework. For example, we define operations to "split" the *first* clause occurring in a goal. We say a goal G is *satisfisfiable* iff the conjunction of the formulas occurring in G is satisfiable. Using ML-like syntax, we define:

$$\begin{array}{ll} \textit{goal} & = \textit{formula sequence} \times \textit{attribute sequence} \end{array}$$

We say a goal is *trivially satisfiable* if the formula sequence is empty, and it is *trivially unsatisfiable* if the formula sequence constains the formula *false*. We say a goal is *basic* if it is trivially satisfiable or unsatisfiable.

Tactics. In our approach, when a tactic is applied to some goal G, four different outcomes are possible:

- The tactic succeeds in showing G to be satisfiable,
- The tactic succeeds in showing G to be unsatisfiable,
- The tactic produces a sequence of subgoals,
- The tactic fails.

A tactic returns a *model* when it shows G to be satisfiable, and a *proof* when it shows G to be unsatisfiable. A model is a sequence of assignments of symbols to values. Such symbols may be constant symbols, function symbols, etc. Values may be booleans, integers, algebraic reals, bit-vectors, lambda terms, etc. An empty model is an empty sequence of assignments. A proof of unsatisfiability may be a full certificate that can be independently checked by a proof checker, or it may be just a subset of the goals (also known as an unsat core) that were used to demonstrate the unsatisfiability. In this paper, we intentionally do not specify how proofs and models are represented; the framework we describe is general enough to accommodate many different decisions in this regard. In fact, **RAHD** and **Z3** use different representations.

When reducing a goal G to a sequence of subgoals $G_1, \ldots, G_n$, we face the problems of proof and model conversion. A *proof converter* constructs a proof of unsatisfiability for G using the proofs of unsatisfiability for all subgoals $G_1, \ldots, G_n$. Analogously, a *model converter* constructs a model for G using a model for some subgoal G_i. In the type declarations below, we use *trt* to stand for "tactic return type."

$$\begin{array}{ll}
\textit{proofconv} & = \textit{proof sequence} \rightarrow \textit{proof} \\
\textit{modelconv} & = \textit{model} \times \textit{nat} \rightarrow \textit{model} \\
\textit{trt} & = \mathsf{sat}\ \textit{model} \\
 & \mid\ \mathsf{unsat}\ \textit{proof} \\
 & \mid\ \mathsf{unknown}\ \textit{goal sequence} \times \textit{modelconv} \times \textit{proofconv} \\
 & \mid\ \mathsf{fail} \\
\textit{tactic} & = \textit{goal} \rightarrow \textit{trt}
\end{array}$$

The second parameter of a model converter is a natural number used to communicate to the model converter which subgoal was shown to be satisfiable.

Let us gain some intuition about tactics and tacticals in the context of SMT with a few simple examples.

The tactic elim eliminates constants whenever the given goal contains equations of the form $a = t$, where a is a constant and t is a term not containing a.

For example, suppose elim is applied to a goal containing the following sequence comprised of three formulas:

$$[\ a = b + 1,\ \ (a < 0 \vee a > 0),\ \ b > 3\ ]$$

The result will be unknown(s, mc, pc), where s is a sequence containing the single subgoal:

$$[\ (b + 1 < 0 \vee b + 1 > 0),\ \ b > 3\ ]$$

The model converter mc is a function s.t. when given a model M for the subgoal above, mc will construct a new model M' equal to M except that the interpretation of a in M' ($M'(a)$) is equal to the interpretation of b in M plus one (i.e., $M(b) + 1$). Similarly, the proof converter pc is a function s.t. given a proof of unsatisfiability for the subgoal will construct a proof of unsatisfiability for the original goal using the fact that $(b + 1 < 0 \vee b + 1 > 0)$ follows from $a = b + 1$ and $(a < 0 \vee a > 0)$.

The tactic split-or splits a disjunction of the form $p_1 \vee \ldots \vee p_n$ into cases and then returns n subgoals. If the disjunction to be split is not specified, the tactic splits the first disjunction occurring in the input goal. For example, given the goal G comprising of the following sequence of formulas:

$$[\ a = b + 1,\ \ (a < 0 \vee a > 0),\ \ b > 3\ ]$$

split-or G returns unknown($[G_1, G_2]$, mc, pc), where G_1 and G_2 are the subgoals comprised of the following two formula sequences respectively:

$$[\ a = b + 1,\ \ a < 0,\ \ b > 3\ ]$$

$$[\ a = b + 1,\ \ a > 0,\ \ b > 3\ ]$$

The model converter mc is just the identity function, since any model for G_1 or G_2 is also a model for G. The proof converter pc just combines the proofs of unsatisfiability for G_1 and G_2 in a straighforward way. If G does not contain a disjunction, then split-or just returns the input goal unmodified. Another option would be to fail.

RAHD and **Z3** come equipped with several built-in tactics. It is beyond the scope of this paper to document all of them. Nevertheless, let us list some of them for illustrative purposes:

- simplify: Apply simplification rules such as constant folding (e.g., $x + 0 \rightsquigarrow x$).
- nnf: Put the formula sequence in negation normal form.
- cnf: Put the formula sequence in conjunctive normal form.
- tseitin: Put the formula sequence in conjunctive normal form, but use fresh Boolean constants and predicates for avoiding exponential blowup. The model converter produced by this tactic “erases” these fresh constants and predicates introduced by it.
- lift-if: Lift term if-then-else’s into formula if-then-else’s
- bitblast: Reduce bitvector terms into propositional logic.

– gb: Let E be the set of arithmetic equalities in a goal G, gb replaces E with the Gröbner basis induced by E.
– vts: Perform virtual term substitution.
– propagate-bounds: Perform bound propagation using inference rules such as $x < 1 \wedge y < x$ implies $y < 1$.
– propagate-values: Perform value propagation using equalities of the form $t = a$ where a is a numeral.
– split-ineqs: Split inequalities such as $t \leq 0$ into $t = 0 \vee t < 0$.
– som: Put all polynomials in sum of monomials form.
– cad: Apply cylindrical algebraic decomposition.

Tacticals. It is our hope[11] that tactics will be made available in the APIs of next generation SMT solvers. Developers of interactive and automated reasoning systems will be able to combine these tactics using their favorite programming language. Like in LCF, it is useful to provide a set of combinators (tacticals) that are used to combine built-in tactics into more complicated ones. The main advantage of using tacticals is that the resulting tactic is guaranteed to be correct, that is, it is sound if the used building blocks are sound, it connects the model converters and proof converters appropriately, and there is no need to keep track of which subgoals were already proved to be unsatisfiable. We propose the following basic tacticals:

then : $(\textit{tactic} \times \textit{tactic}) \rightarrow \textit{tactic}$

then(t_1, t_2) applies t_1 to the given goal and t_2 to every subgoal produced by t_1. The resulting tactic fails if t_1 fails when applied to the goal, or if t_2 does when applied to any of the resulting subgoals.

then$*$: $(\textit{tactic} \times \textit{tactic sequence}) \rightarrow \textit{tactic}$

then$*(t_1, [t_{2_1}, ..., t_{2_n}])$ applies t_1 to the given goal, producing subgoals $g_1, ..., g_m$. If $n \neq m$, the tactic fails. Otherwise, it applies, in parallel, t_{2_i} to every goal g_i. The resultant set of subgoals is the union of all subgoals produced by t_{2_i}'s. The resulting tactic fails if t_1 fails when applied to the goal, or if t_{2_i} does when applied to goal g_i. The resultant tactic also fails if the number of subgoals produced by t_1 is not n.

orelse : $(\textit{tactic} \times \textit{tactic}) \rightarrow \textit{tactic}$

orelse(t_1, t_2) first applies t_1 to the given goal, if it fails then returns the result of t_2 applied to the given goal.

par : $(\textit{tactic} \times \textit{tactic}) \rightarrow \textit{tactic}$

par(t_1, t_2) executes t_1 and t_2 in parallel to the given goal. The result is the one produced by the first tactic to terminate its execution. After one tactic terminates, the execution of the other one is terminated. The resulting tactic fails only if t_1 and t_2 fails when applied to the goal.

[11] In fact, **Z3** 4.0 is now available with all of the strategy machinery described in this paper. It uses the strategy language internally and publishes a strategy API. Bindings of the strategy API are also available within Python. This Python **Z3** strategy interface can be experimented with on the web at `http://rise4fun.com/Z3Py`.

repeat : $\mathit{tactic} \rightarrow \mathit{tactic}$

Keep applying the given tactic until no subgoal is modified by it. repeat(t) fails if t fails.

repeatupto : $\mathit{tactic} \times \mathit{nat} \rightarrow \mathit{tactic}$

Keep applying the given tactic until no subgoal is modified by it, or the maximum number of iterations is reached. repeatupto(t) fails if t fails.

tryfor : $\mathit{tactic} \times \mathit{milliseconds} \rightarrow \mathit{tactic}$

tryfor(t, k) returns the value computed by tactic t applied to the given goal if this value is computed within k milliseconds, otherwise it fails. The resulting tactic also fails if t fails when applied to the goal.

The tactic skip never fails and just returns the input goal unaltered, it is the unit for then: $\mathsf{then}(\mathsf{skip}, t) = \mathsf{then}(t, \mathsf{skip}) = t$; and fail always fails, and is the unit for orelse: $\mathsf{orelse}(\mathsf{fail}, t) = \mathsf{orelse}(t, \mathsf{fail}) = t$. Note that then, orelse and par are associative. From now on, in order to simplify the presentation of examples, we write $t_1 \,;\, t_2$ to denote $\mathsf{then}(t_1, t_2)$, and $t_1 \,|\, t_2$ to denote $\mathsf{orelse}(t_1, t_2)$.

Formula Measures. Several SMT solvers use hard-coded strategies that perform different reasoning techniques depending on structural features of the formula being analyzed. For example, **Yices** [21] checks whether a formula is in the difference logic fragment or not. A formula is in the difference logic fragment if all atoms are of the form $x - y \bowtie k$, where x and y are uninterpreted constants, k a numeral, and $\bowtie$ is in $\{\leq, \geq, =\}$. If the formula is in the difference logic fragment, **Yices** checks if the number of inequalities divided by the number of uninterpreted constants is smaller than a threshold k. If this is the case, it uses the Simplex algorithm for processing the arithmetic atoms. Otherwise, it uses an algorithm based on the Floyd-Warshall all-pairs shortest distance algorithm. We call such structural features *formula measures*. This type of ad hoc heuristic strategy based upon formula measures is very common.

We use formula measures to create Boolean expressions that are evaluated over goals. The built-in tactic check : $\mathit{cond} \rightarrow \mathit{tactic}$ fails if the given goal does not satisfy the condition *cond*, otherwise it just returns the input goal. Many numeric and Boolean measures are available in **RAHD** and **Z3**. Here is an incomplete list for illustrative purposes:

bw: Sum total bit-width of all rational coefficients of polynomials.
diff: True if the formula is in the difference logic fragment.
linear: True if all polynomials are linear.
dim: Number of uninterpreted constants (of sort real or int).
atoms: Number of atoms.
degree: Maximal total multivariate degree of polynomials.
size: Total formula size.

Nontrivial conditions can be defined using the built-in measures, arithmetic operators (e.g., $+$, $-$, $\times$, $/$, $<$, $=$) and Boolean operators (e.g., $\wedge$, $\vee$, $\neg$). For example, the **Yices** strategy described above can be encoded as:

$$(\mathsf{check}(\neg\mathsf{diff} \vee \frac{\mathsf{atom}}{\mathsf{dim}} < \mathsf{k})\,;\,\mathsf{simplex})\,|\,\mathsf{floydwarshall}$$

Now, we define the combinators if and when based on the combinators and tactics defined so far.

$$\begin{aligned} \mathsf{if}(c,\ t_1,\ t_2) &= (\mathsf{check}(c)\,;\,t_1)\,|\,t_2 \\ \mathsf{when}(c,\ t) &= \mathsf{if}(c,\ t,\ \mathsf{skip}) \end{aligned}$$

These are often helpful in the construction of strategies based upon formula measures.

Under and over-approximations. Under and over-approximation steps are commonly used in SMT solvers. An *under-approximation step* consists of reducing a set of formulas S to a set S' such that if S' is satisfiable, then so is S, but if S' is unsatisfiable, then nothing can be said about S. For example, any *strengthening* step that obtains S' by adding to S new formulas not deducible from S is an under-approximation.

A more concrete example is found in many SMT solvers for nonlinear integer arithmetic, where lower and upper bounds are added for every uninterpreted constant of sort int, and the resulting set of formulas is then reduced to SAT. Under-approximations are also used in finite model finders for first-order logic formulas, where the universe is assumed to be finite, and the first-order formula is then reduced into SAT. Analogously, an *over-approximation step* consists of reducing a set of formulas S into a set S' such that if S' is unsatisfiable, then so is S, but if S' is satisfiable, then nothing can be said about S. For example, any *weakening* step that removes formulas from S is an over-approximation. Boolean abstraction is another example used in many interactive theorem provers and SMT solvers. This comprises replacing every theory atom with a fresh propositional variable. Clearly, if the resulting set of formulas is unsatisfiable then so is the original set. Of course, given a set of formulas S, arbitrarily applying under and over-approximation steps result in set of formulas S' that cannot be used to establish the satisfiability nor the unsatisfiability of S. To prevent under and over-approximation steps from being incorrectly applied, we associate a precision attribute with every goal. A precision marker is an element of the set $\{\mathsf{prec}, \mathsf{under}, \mathsf{over}\}$. A tactic that applies an under (over) approximation fails if the precision attribute of the input goal is over (under).

4 Parametric Reasoning Engines as Tacticals

Some reasoning engines utilize other engines as subroutines. It is natural to view these higher-level reasoning engines as tacticals. Given a subsidiary engine (a tactic given to the higher-level engine as a parameter), these tacticals produce a new tactic. Let us describe two examples of such parametric engines.

SMT solvers. We observe three main phases in state-of-the-art SMT solvers: *preprocessing*, *search*, and *final check*.

Preprocessing During preprocessing, also known as presolving, several simplifications and rewriting steps are applied. The main goal is to put the problem in a form that is suitable for solving. Another objective is to simplify the problem, eliminate uninterpreted constants, unconstrained terms, and redundancies. Some solvers may also apply reduction techniques such as bit-blasting where bit-vector terms are reduced to propositional logic. Another commonly used reduction technique is Ackermannization [2,12] where uninterpreted function symbols are eliminated at the expense of introducing fresh constants and additional constraints.

Search During the search step, modern SMT solvers combine efficient SAT solving with "cheap" theory propagation techniques. Usually, this combination is an incomplete procedure. For example, consider problems containing arithmetic expressions. Most solvers ignore integrality and nonlinear constraints during the search step. These solvers will only propagate Boolean and linear constraints, and check whether there is a rational assignment that satisfies them. We say the solver is *postponing* the application of "expensive" and complete procedures to the final check step. Solvers, such as **Z3**, only process nonlinear constraints during final check. The word "final" is misleading since it may be executed many times for a give problem. For example, consider the following nonlinear problem comprising of three assumptions (over $\mathbb{R}$):

$$[\ x = 1,\ y \geq x + 1,\ (y \times y < 1 \vee y < 3 \vee y \times y > x + 3)\ ]$$

In the preprocessing step, a solver may decide to eliminate x using Gaussian elimination obtaining:

$$[\ y \geq 2,\ (y \times y < 1 \vee y < 3 \vee y \times y > 4)\ ]$$

During the search step, the solver performs only Boolean propagation and cheap theory propagation such as $y \geq 2$ implies $\neg(y < 2)$. Nonlinear monomials, such as $y \times y$, are treated as fresh uninterpreted constants. Thus, the incomplete solver used during the search may find the candidate assigment $y = 2$ and $y \times y = 0$. This assignment satisfies the atoms $y \geq 2$ and $y \times y < 1$, and all Boolean and linear constraints.

Final check During final check, a complete procedure for nonlinear real arithmetic is used to decide $[\ y \geq 2,\ y \times y < 1\ ]$. The complete procedure finds it to be unsatisfiable, and the solver backtracks and learns the lemma $(\neg y \geq 2 \vee y \times y < 1)$. The search step resumes, and finds a new assignment that satisfies $[\ y \geq 2,\ y \times y > 4\ ]$. The final check step is invoked again, and this time it finds the constraints to be satisfiable and the SMT solver terminates. The procedure above can be encoded as tactic of the form:

$$\mathsf{preprocess}\,;\,\mathsf{smt}(\mathsf{finalcheck})$$

where preprocess is a tactic corresponding to the preprocessing step, and finalcheck is another tactic corresponding to the final check step, and smt is a tactical. The smt tactical uses a potentially expensive finalcheck tactic to complement an incomplete and fast procedure based on SAT solving and cheap theory propagation.

Abstract Partial Cylindrical Algebraic Decomposition (AP-CAD). AP-CAD [36,38] is an extension of the well-known real closed field (RCF) quantifier elimination procedure *partial cylindrical algebraic decomposition* (P-CAD). In AP-CAD, arbitrary (sound but possibly incomplete) ∃-RCF decision procedures can be given as parameters and used to "short-circuit" certain expensive computations performed during CAD construction. The ∃-RCF decision procedures may be used to reduce the expense of the different phases of the P-CAD procedure. The key idea is to use some *fast*, *sound* and *incomplete* procedure P to improve the performance of a *complete* but potentially very expensive procedure. The procedure P may be the combination of different techniques based on interval constraint propagation, rewriting, Gröbner basis computation, to cite a few. These combinations may be tailored as needed for different application domains. These observations suggest that P is a tactic, and AP-CAD is tactical that given P returns a tactic that implements a complete ∃-RCF decision procedure.

We now illustrate the flexibility of our approach using the following simple strategy for nonlinear arithmetic:

$$\mathsf{simplify}\,;\,\mathsf{gaussian}\,;\,(\mathsf{modelfinder}\,|\,\mathsf{smt}(\mathsf{apcad}(\mathsf{icp})))$$

The preprocessing step is comprised of two steps. First, simple rewriting rules such as constant folding and gaussian elimination are applied. Then, a model finder for nonlinear arithmetic based on SAT [49] is used. If it fails, smt is invoked using AP-CAD (apcad) in the final check step. Finally, AP-CAD uses interval constraint propagation (icp) to speedup the AP-CAD procedure.

5 Strategies in Action

We demonstrate the practical value of our approach by describing successful strategies used in **RAHD** and **Z3**. We also provide evidence that the overhead due to the use of tactics and tacticals is insignificant, and the gains in performance substantial.

5.1 Z3 QF_LIA strategy

SMT-LIB [4] is a repository of SMT benchmark problems. The benchmarks are divided in different divisions. The QF_LIA division consists of linear integer arithmetic benchmarks. These benchmarks come from different application domains such as scheduling, hardware verification, software analysis and bounded-model checking. The structural characteristics of these problems are quite diverse. Some of them contain a rich Boolean structure, and others are just the conjunction of linear equalities and inequalities. Several software analysis benchmark make extensive use of if-then-else terms that need to be eliminated during a preprocessing step. A substantial subset of the benchmarks are unsatisfiable even when integrality constraints are ignored, and can be solved using a procedure for linear real arithmetic, such as Simplex. We say a QF_LIA benchmark is *bounded* if every uninterpreted constant a has a lower ($k \leq a$) and upper bound ($a \leq k$), where k is a numeral. A bounded benchmark is said to be 0-1 (or *pseudo-boolean*) if the lower (upper) bound of every uninterpreted constant is 0 (1). Moreover, some of the problems in QF_LIA become bounded after interval constraint propagation is applied.

Z3 3.0 won the QF_LIA division in the 2011 SMT competition[12] (SMT-COMP'11). The strategy used by **Z3** can be summarized by the following tactic:

$\mathsf{preamble}\,;\,(\mathsf{mf} \mid \mathsf{pb} \mid \mathsf{bounded} \mid \mathsf{smt})$

where the preamble, mf, pb and bounded tactics are defined as

$$
\begin{aligned}
\mathsf{preamble} &= \mathsf{simplify}\,;\,\mathsf{propagate\text{-}values}\,;\,\mathsf{ctx\text{-}simplify}\,;\,\mathsf{lift\text{-}if}\,;\,\mathsf{gaussian}\,;\,\mathsf{simplify}\\
\mathsf{mf} &= \mathsf{check}(\mathsf{is\text{-}ilp})\,;\,\mathsf{propagate\text{-}bounds}\,;\\
&\quad \begin{pmatrix} \mathsf{tryfor}(\mathsf{mip}, 5000) & \mid \mathsf{tryfor}(\mathsf{smt\text{-}no\text{-}cut}(100), 2000) \mid \\ (\mathsf{add\text{-}bounds}(-16, 15)\,;\,\mathsf{smt}) & \mid \mathsf{tryfor}(\mathsf{smt\text{-}no\text{-}cut}(200), 5000) \mid \\ (\mathsf{add\text{-}bounds}(-32, 31)\,;\,\mathsf{smt}) & \mid \mathsf{mip} \end{pmatrix}\\
\mathsf{pb} &= \mathsf{check}(\mathsf{is\text{-}pb})\,;\,\mathsf{pb2bv}\,;\,\mathsf{bv2sat}\,;\,\mathsf{sat}\\
\mathsf{bounded} &= \mathsf{check}(\mathsf{bounded})\,;\,\begin{pmatrix} \mathsf{tryfor}(\mathsf{smt\text{-}no\text{-}cut}(200), 5000) \mid \\ \mathsf{tryfor}(\mathsf{smt\text{-}no\text{-}cut\text{-}no\text{-}relevancy}(200), 5000) \mid \\ \mathsf{tryfor}(\mathsf{smt\text{-}no\text{-}cut}(300), 15000) \end{pmatrix}
\end{aligned}
$$

The tactic smt is based on the **Yices** approach for linear integer arithmetic. The tactic ctx-simplify performs contextual simplification using rules such as:

$$(a \neq t \vee F[a]) \;\rightsquigarrow\; (a \neq t \vee F[t])$$

The tactic mip implements a solver for mixed integer programming. It can only process conjunctions of linear equations and inequalities. The tactic fails if the input goal contains other Boolean connectives. The tactic smt-no-cut(seed) is a variation of the **Yices** approach in which Gomory cuts are not used. The parameter seed is a seed for the pseudo-random number generator. It is used to randomize the search. The tactic smt-no-cut-no-relevancy(seed) is yet another variation where "don't care" propagation is disabled [17]. The tactic pb2bv converts a pseudo-boolean formula into a bit-vector formula. It fails if the input goal is not pseudo-boolean. Similarly, the tactic bv2sat bitblasts bit-vector terms into propositional logic. The tactic sat implements a SAT solver. Finally, the tactic add-bounds(lower, upper) performs an under-approximation by adding lower and upper bounds to all uninterpreted integer constants. The idea is to guarantee that the branch-and-bound procedure used in smt and mip terminates. The tactic mf is essentially looking for models where all integer variables are assigned to

[12] http://www.smtcomp.org

small values. The tactic pb is a specialized 0-1 (Pseudo-Boolean) solver. It fails if the problem is not 0-1.

To demonstrate the benefits of our approach we run all QF_LIA benchmarks using the following variations of the strategy above:

pre = preamble ; smt

pre+pb = preamble ; (pb | smt)

pre+bounded = preamble ; (bounded | smt)

pre+mf = preamble ; (mf | smt)

combined = preamble ; (mf | pb | bounded | smt)

All experiments were conducted on an Intel Quad-Xeon (E54xx) processor, with individual runs limited to 2GB of memory and 600 seconds. The results of our experimental evaluation are presented in Table 1. The rows are associated with the individual benchmark families from QF_LIA division, and columns separate different strategies. For each benchmark family we write the number of benchmarks that each strategy failed to solve within the time limit, and the cumulative time for the solved benchmarks.

Table 1. Detailed Experimental Results.

	smt		pre		pre+pb		pre+bounded		pre+mf		combined	
benchmark family	failed	time (s)	failed	time (s)	failed	time (s)	failed	time (s)	failed	time (s)	failed	time (s)
Averest (19)	0	4.0	0	5.9	0	6.0	0	5.9	0	5.9	0	5.9
bofill sched (652)	1	1530.7	1	1208.3	1	1191.5	1	1205.4	1	1206.0	1	1205.9
calypto (41)	1	2.0	1	7.7	1	8.0	1	7.8	1	7.9	1	7.8
CAV 2009 (600)	190	1315.3	190	1339.3	190	1329.7	190	1342.7	1	8309.5	1	8208.1
check (5)	0	0.1	0	0.1	0	0.1	0	0.1	0	0.1	0	0.1
CIRC (51)	17	188.4	17	239.8	17	238.2	17	336.1	17	221.5	8	158.66
convert (319)	206	1350.5	190	3060.6	190	3025.9	0	112.7	190	3030.2	0	112.5
cut lemmas (100)	48	2504.0	48	2532.4	48	2509.2	48	2543.4	27	3783.9	27	3709.0
dillig (251)	68	1212.0	68	1237.6	68	1226.9	68	1242.5	3	2677.8	3	2763.9
mathsat (121)	0	171.4	0	150.2	0	149.9	0	151.1	0	150.9	0	150.2
miplib2003 (16)	5	53.8	5	57.7	5	424.4	5	109.5	5	58.8	5	430.5
nec-smt (2780)	147	224149.0	8	59977.4	8	59968.3	8	59929.3	8	60042.1	8	60032.9
pb2010 (81)	43	90.3	43	96.2	25	2581.2	43	146.3	43	96.2	25	2583.1
pidgeons (19)	0	0.3	0	0.4	0	0.4	0	0.3	0	0.3	0	0.3
prime-cone (37)	13	9.6	13	9.5	13	9.5	13	9.7	0	11.0	0	11.0
rings (294)	48	4994.4	46	5973.7	46	6016.2	48	9690.0	46	6024.6	48	9548.2
rings pre (294)	57	441.5	54	1288.7	54	1261.9	54	1260.9	54	1274.7	54	1261.5
RTCL (2)	0	0.1	0	0.1	0	0.1	0	0.1	0	0.1	0	0.1
slacks (251)	135	1132.9	136	550.0	136	545.9	136	550.8	53	8969.3	53	8803.9
total (5938)	978	239153.0	819	77737.4	801	80495.2	631	78646.5	449	95872.4	234	98995.2

Overall, the combined strategy is the most effective one, solving the most problems. It fails only on 234 out of 5938 benchmarks. In constrast, the basic smt strategy fails on 978 benchmarks. The results also show which tactics are effective in particular benchmark families. The tactics ctx-simplify and lift-if are particularly effective on the NEC software verification benchmarks (sf nec-smt). The pseudo-boolean strategy reduces by half the number of failures in the industrial pseudo-boolean benchmarks coming from the 2010 pseudo-boolean competition. The convert software verification benchmarks become trivial when Gomory cuts are disabled by the tactic bounded. Finally, the model finder tactic mf is very effective on crafted benchmark families such as CAV 2009, cut lemmas, dillig, prime-cone, and slacks.

Figure 1 contains scatter plots comparing the strategies described above. Each point on the plots represents a benchmark. The plots are in log scale. Points below (above) the diagonal are benchmarks where the strategy on y-axis (x-axis) is faster than the strategy on the x-axis (y-axis). Note that in some cases, the combined strategy has a negative impact, but it overall solves more problems.

We observed several benefits when using tactics and tacticals in **Z3**. First, it is straighforward to create complex strategies using different solvers and techniques. The different solvers can be implemented and maintained independently of each other. The overhead of using tactics and tacticals is insignificant. We can provide custom strategies to **Z3** users in different application domains. Finally, the number of SMT-LIB problems that could be solved by **Z3** increased dramatically. **Z3** 2.19 uses only the (default) smt tactic, and fails to solve 978 (out of 5938) QF_LIA benchmarks with a 10 minutes timeout. In contrast, **Z3** 3.0 fails in only 234 benchmarks. In **Z3** 4.0, tactic and tacticals are available in the programmatic API and SMT 2.0 frontend.

5.2 Calculemus RAHD strategies

The calculemus **RAHD** strategies[13] combine simplification procedures, interval constraint propagation, Gröbner basis computation, non-strict inequality splitting, DNF conversion, OpenCAD and CAD. OpenCAD is a variation of the CAD procedure that can be applied to problems containing $\wedge\vee$ combinations of polynomial strict inequalities. OpenCAD takes advantage of the topological structure of the solution set of these formulas (such solution sets are always open subsets of $\mathbb{R}^n$) to yield a proof procedure substantially faster than general CAD. This speed-up is caused by (i) the use rational numbers instead of irrational real algebraic numbers to represent CAD sample points, and (ii) the use of an efficient projection operator which avoids the costly computation of polynomial subresultants.

The key idea of the calculemus strategy is to split non-strict inequalities ($p \leq 0$) appearing in a conjunctive formula F into ($p < 0 \vee p = 0$), resulting in two sub-problems $F_<$ and $F_=$. The branch $F_<$ containing the strict inequality is then closer to being able to be processed using OpenCAD, while the branch $F_=$ containing the equality has an enriched equational structure which is then used, via Gröbner basis computation, to inject equational information into the polynomials appearing in the strict inequalities in $F_=$. If the ideal generated by the equations in the branch $F_=$ is rich enough and the original formula is unsatisfiable, then this unsatisfiability of $F_=$ may be recognized by applying

[13] Detailed descriptions of these strategies may be found in Passmore's PhD thesis [36].

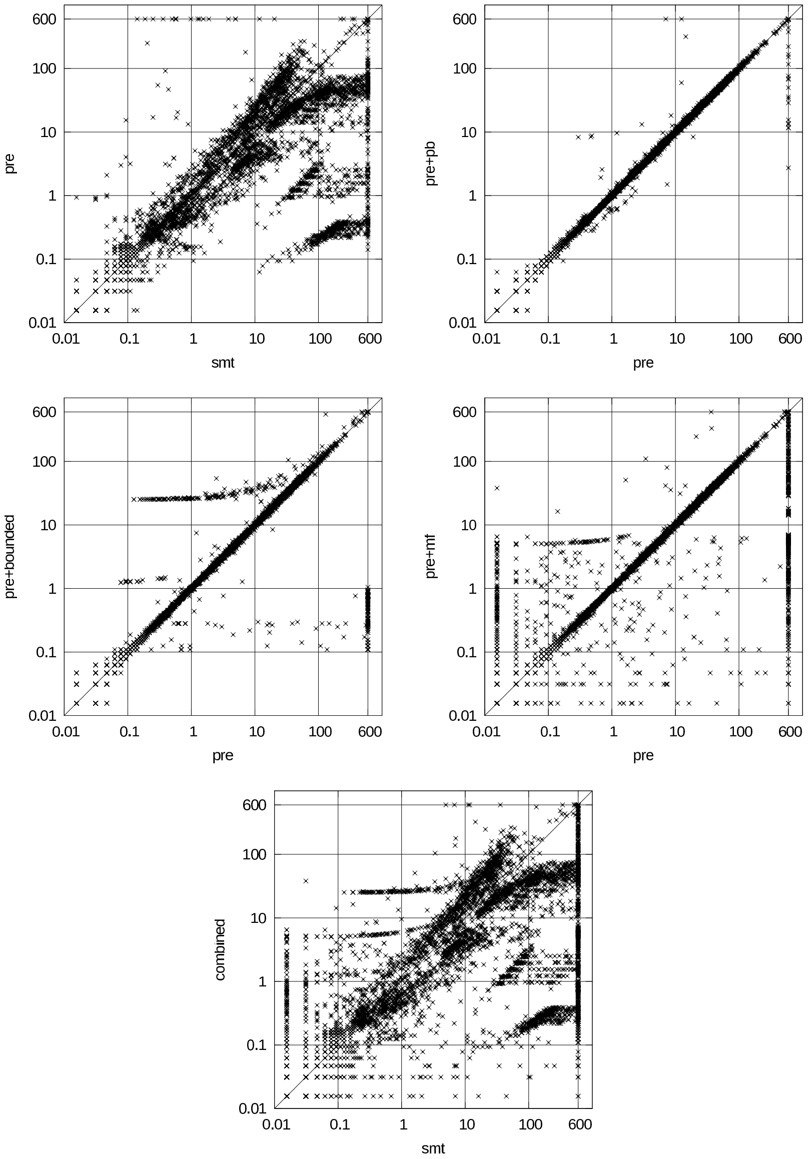

Fig. 1. smt, pre, pre+pb, pre+bounded, pre+mf and combined strategies.

OpenCAD only to the resulting strict inequational fragment of $F_=$ after this Gröbner basis reduction has been performed.

In this section, we consider the basic calculemus strategy calc-0, and two refinements: calc-1 and calc-2. These refinements use formula measures to control inequality splitting. Moreover, interval constraint propagation is used to close goals before further splitting is performed.

Table 2. The three **RAHD** `calculemus` proof strategies compared with QEPCAD-B and Redlog on twenty-four problems.

			calc-0	calc-1	calc-2	qepcad-b	redlog/rlqe	redlog/rlcad
benchmark	dimension	degree	time (s)	time (s)	time (s)	time (s)	time (s)	time (s)
P0	5	4	0.9	1.6	1.7	416.4	40.4	>600.0
P1	6	4	1.7	3.1	3.4	>600.0	>600.0	>600.0
P2	5	4	1.3	2.4	2.6	>600.0	>600.0	>600.0
P3	5	4	1.5	2.5	2.7	>600.0	>600.0	>600.0
P4	5	4	1.1	2.0	2.7	>600.0	>600.0	>600.0
P5	14	2	0.3	0.3	0.3	>600.0	97.4	>600.0
P6	11	5	147.4	<0.1	<0.1	>600.0	<0.1	<0.1
P7	8	2	<0.1	<0.1	<0.1	<0.1	<0.1	<0.1
P8	7	32	4.5	0.1	<0.1	8.4	<0.1	>600.0
P9	7	16	4.5	0.2	<0.1	0.3	<0.1	6.7
P10	7	12	100.7	20.8	8.9	>600.0	>600.0	>600.0
P11	6	2	1.6	0.5	0.5	<0.1	<0.1	<0.1
P12	5	3	0.8	0.3	0.4	<0.1	<0.1	<0.1
P13	4	10	3.8	3.9	4.0	>600.0	>600.0	>600.0
P14	2	2	4.5	1.7	<0.1	<0.1	>600.0	>600.0
P15	4	3	0.2	0.2	0.1	<0.1	<0.1	<0.1
P16	4	2	10.0	2.2	2.1	<0.1	<0.1	<0.1
P17	4	2	0.6	0.6	0.7	0.3	<0.1	0.6
P18	4	2	1.3	1.3	1.3	<0.1	<0.1	<0.1
P19	3	6	3.3	1.7	2.1	<0.1	<0.1	0.7
P20	3	4	1.2	0.7	0.7	<0.1	<0.1	0.3
P21	3	2	<0.1	<0.1	<0.1	<0.1	<0.1	<0.1
P22	2	4	<0.1	<0.1	<0.1	<0.1	<0.1	<0.1
P23	2	2	<0.1	<0.1	<0.1	<0.1	<0.1	<0.1

Table 2 shows the performance of the calculemus **RAHD** strategies on the twenty-four benchmarks considered in [37] and compares this performance to that of QEPCAD-B [11] and two quantifier elimination procedures available in Reduce/Redlog [1]:

– Rlqe, which is an enhanced implementation by Dolzmann and Sturm of Weispfenning's quadratic virtual term substitution (VTS) [46] , and
– Rlcad, which is an implementation by Seidl, Dolzmann and Sturm of Collins-Hong's partial CAD [20].

Comparing these calculemus strategies with each other and with other tools (QEPCAD-B, Redlog/rlqe, Redlog/rlcad) is interesting in this setting, as in addition to having its own implementation of many real algebraic decision methods, **RAHD** also provides access to these other tools as tactics. For instance, the OpenCAD tactic used in the calculemus strategies is actually executed by **RAHD** invoking QEPCAD-B in a special mode which only lifts over full-dimensional cells. Thus, the comparison given in **Table 2** is indeed a comparison between six **RAHD** strategies: the three calculemus strategies, and three simple strategies which only invoke QEPCAD-B, Redlog/Rlqe and Redlog/Rlcad, respectively.

For each benchmark we write the dimension and maximal total multivariate degree of polynomials, and the total runtime for each strategy and solver. Experiments were performed on a 2 x 2.4 GHz Quad-Core Intel Xeon PowerMac with 10GB of 1066 MHz DDR3 RAM.

For brevity, let us only compare `calc-0` with the QEPCAD-B and Redlog procedures. With this restriction, the results of these experiments can be broadly summarized as follows:

- The `calc-0` strategy is able to solve a number of high-dimension, high-degree problems that QEPCAD-B, Redlog/Rlqe, and Redlog/Rlcad are not. It is interesting that while the `calc-0` strategy involves an exponential blow-up in its reliance on inequality splitting followed by a DNF normalisation, for many benchmarks the increase in complexity caused by this blow-up is overshadowed by the decrease in complexity of the CAD-related computations this process induces.
- Redlog/Rlqe is able to solve a number of high-dimension, high-degree benchmarks that QEPCAD-B and Redlog/Rlcad are not.
- Redlog/Rlqe is able to solve a number of benchmarks significantly faster than the `calc-0` strategy, Redlog/Rlcad, and QEPCAD-B.
- For the benchmarks QEPCAD-B is able to solve directly, using QEPCAD-B directly tends to be much faster than using the `calc-0` strategy.

Overall, the final refinement, `calc-2`, substantially improves upon the strategy `calc-0` on benchmarks P6, P8, P10, P11, P12, P14, P16, P19 and P20, often by many orders of magnitude. On benchmarks P0, P1, P2, P3, P4, `calculemus-2` is slower than `calc-0` by roughly a factor of two. Strategies `calc-1` and `calc-2` are roughly equal for most benchmarks, except for P1 and P19 where `calc-2` is slightly ($\cong$ 10-20%) slower, and P10 and P14 where `calc-2` is substantially ($\cong$ 2-25x) faster.

6 Conclusion

We have demonstrated the practical value of heuristic proof strategies within the context of our **RAHD** and **Z3** tools. We have illustrated that not only is a strategy-language based approach practical in the context of high-performance solvers, it is also desirable. A key take-away message is the following: In difficult (i.e., infeasible or undecidable) theorem proving domains, the situation with heuristic proof strategies is rarely "one size fits all." Instead, given a class of problems to solve, it is often the case that one heuristic combination of reasoning engines is far more suited to the task than another. SMT solver developers cannot anticipate all classes of problems end-users will wish to analyze. By virtue of this, heuristic components of high-performance solvers will never be sufficient in general when they are beyond end-users' control. Without providing end-users mechanisms to control and modify the heuristic components of their solvers, solver developers are inhibiting their chances of success.

Beyond the situation with end-users, let us also make the following anecdotal remarks as solver developers. By introducing a strategy language foundation into our solvers, we have found our productivity radically enhanced, especially when faced with the goal of solving new classes of problems. The strategy language framework allows us to easily modify and experiment with variations of our solving heuristics. Before we had such strategy language machinery in place, with its principled handling of goals, models and proofs, this type of experimentation with new heuristics was cumbersome and error-prone.

We have proposed a Strategy Challenge to the SMT community: To build theoretical and practical tools allowing users to exert strategic control over core heuristic aspects of high-performance SMT solvers. We discussed some of the rich history of ideas of strategy in the context of mechanized proof, and presented an SMT-oriented approach for orchestrating reasoning engines, where "big" symbolic reasoning steps are represented as tactics, and these tactics are composed using combinators known as tacticals. We demonstrated the practical value of this approach by describing a few examples of how tactics and tacticals have been successfully used in our **RAHD** and **Z3** tools.

There are several directions for future work. First, we believe that many other authors of SMT solvers should take up this Strategy Challenge, and much experimentation should be done — from many different points of view and domains of application — before a standard strategy language for SMT can be proposed. When the time is right, we believe that the existence of a strategy standard (extending, for instance, the SMT-LIB standard) and the development and study of theoretical frameworks for SMT strategies could give rise to much progress in the practical efficacy of automated reasoning tools.

Second, we would like to understand how one might *efficiently* exert "small step" strategic control over reasoning engines. Abstract proof procedures, such as Abstract DPLL [35], DPLL(T) [35] and cutsat [26], represent a proof procedure as a set of transition rules. In these cases, a strategy comprises a *recipe* for applying these "small" step rules. Actual implementations of these abstract procedures use carefully chosen efficient data-structures that depend on the preselected strategy. It is not clear to us how to specify a strategy for these abstract procedures so that an efficient implementation can be automatically generated. Another topic for future investigation is to explore different variations of the LCF approach, such as the ones used by the interactive theorem provers Isabelle [40], HOL [24], Coq [6], MetaPRL [25] and Matita [3].

Acknowledgements. We thank Rob Arthan, Jeremy Avigad, Maria Paola Bonacina, Bob Boyer, Barbara Galletly, Paul B. Jackson, Joost Joosten and Larry Paulson for helpful comments on a draft of this paper. We thank the referees for their many useful corrections and suggestions. Grant Passmore was supported during this research by the UK Engineering and Physical Sciences Research Council [grant numbers EP/I011005/1 and EP/I010335/1].

References

1. T. S. A. Dolzmann. Redlog User Manual - Edition 2.0. MIP-9905, 1999.
2. W. Ackermann. Solvable cases of the decision problem. *Studies in Logic and the Foundation of Mathematics*, 1954.
3. A. Asperti, W. Ricciotti, C. Sacerdoti Coen, and E. Tassi. The Matita Interactive Theorem Prover. In N. Bjørner and V. Sofronie-Stokkermans, editors, *Automated Deduction – CADE-23*, volume 6803 of *Lecture Notes in Computer Science*, pages 64–69. Springer Berlin Heidelberg, 2011.
4. C. Barrett, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.
5. C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany.
6. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer Verlag, 2004.
7. M. Bonacina. A Taxonomy of Theorem-Proving Strategies. In M. Wooldridge and M. Veloso, editors, *Artificial Intelligence Today*, volume 1600 of *Lecture Notes in Computer Science*, pages 43–84. Springer Berlin Heidelberg, 1999.
8. M. P. Bonacina and J. Hsiang. On the modeling of search in theorem proving–towards a theory of strategy analysis. *Inf. Comput.*, 147(2):171–208, Dec. 1998.
9. M. P. Bonacina and W. W. McCune. Distributed theorem proving by Peers. In A. Bundy, editor, *Automated Deduction — CADE-12*, volume 814 of *Lecture Notes in Computer Science*, pages 841–845. Springer Berlin Heidelberg, 1994.
10. M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. Rossum, S. Schulz, and R. Sebastiani. MathSAT: Tight Integration of SAT and Mathematical Decision Procedures. *J. Autom. Reason.*, 35(1-3):265–293, Oct. 2005.
11. C. W. Brown. QEPCAD-B: A System for Computing with Semi-algebraic Sets via Cylindrical Algebraic Decomposition. *SIGSAM Bull.*, 38:23–24, March 2004.
12. R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, A. Santuari, and R. Sebastiani. To Ackermann-ize or Not to Ackermann-ize? On Efficiently Handling Uninterpreted Function Symbols in *UF(E)*. In *LPAR*, pages 557–571, 2006.
13. M. Davis. Eliminating the Irrelevant from Mechanical Proofs. *Proc. Symp. Applied Math.*, XV:15–30, 1963.
14. M. Davis. The early history of automated deduction. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 3–15. Elsevier and MIT Press, 2001.
15. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5:394–397, July 1962.
16. M. Davis and H. Putnam. A computing procedure for quantification theory. *J. ACM*, 7:201–215, July 1960.
17. L. de Moura and N. Bjørner. Relevancy propagation. Technical Report MSR-TR-2007-140, Microsoft Research, 2007.

18. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. of TACAS*, number 4963 in LNCS. Springer, 2008.
19. L. M. de Moura and N. Bjørner. Satisfiability modulo theories: introduction and applications. *Commun. ACM*, 54(9):69–77, 2011.
20. A. Dolzmann, A. Seidl, and T. Sturm. Efficient Projection Orders for CAD. In *ISSAC '04: Proceedings of the 2004 international symposium on Symbolic and algebraic computation*, pages 111–118. ACM, 2004.
21. B. Dutertre and L. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *CAV*, pages 81–94, 2006.
22. P. C. Gilmore. A Proof Method for Quantification Theory: its Justification and Realization. *IBM J. Res. Dev.*, 4:28–35, January 1960.
23. M. Gordon. *From LCF to HOL: a short history*, pages 169–185. MIT Press, Cambridge, MA, USA, 2000.
24. M. J. C. Gordon and T. F. Melham. *Introduction to HOL: a theorem-proving environment for higher-order logic*. Cambridge University Press, 1993.
25. J. J. Hickey. *The MetaPRL Logical Programming Environment*. PhD thesis, Cornell University, Ithaca, NY, January 2001.
26. D. Jovanovic and L. de Moura. Cutting to the chase solving linear integer arithmetic. In *CADE*, pages 338–353, 2011.
27. F. Kirchner and C. Muñoz. The proof monad. *Journal of Logic and Algebraic Programming*, 79(3–5):264–277, 2010.
28. R. Kowalski. Search Strategies for Theorem Proving. *Machine Intelligence*, 5:181–201, 1969.
29. E. L. Lusk. Controlling Redundancy in Large Search Spaces: Argonne-Style Theorem Proving Through the Years. In *Proceedings of the International Conference on Logic Programming and Automated Reasoning*, LPAR '92, pages 96–106, London, UK, UK, 1992. Springer-Verlag.
30. B. Luttik and E. Visser. Specification of rewriting strategies. In M. P. A. Sellink, editor, *2nd International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF 1997)*, Electronic Workshops in Computing, Berlin, November 1997. Springer-Verlag.
31. W. McCune. Solution of the Robbins Problem. *J. Autom. Reason.*, 19(3):263–276, Dec. 1997.
32. W. McCune. Prover9 and Mace4. `http://www.cs.unm.edu/~mccune/prover9/`, 2005–2010.
33. R. Milner. Logic for computable functions: description of a machine implementation. Technical Report STAN-CS-72-288, Stanford University, 1972.
34. A. Newell, J. C. Shaw, and H. A. Simon. Elements of a Theory of Human Problem Solving. *Psychological Review*, 65:151–166, 1958.
35. R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. ACM*, 53(6):937–977, 2006.
36. G. O. Passmore. *Combined Decision Procedures for Nonlinear Arithmetics, Real and Complex*. PhD thesis, University of Edinburgh, 2011.
37. G. O. Passmore and P. B. Jackson. Combined Decision Techniques for the Existential Theory of the Reals. In *Proceedings of the 16th Symposium, 8th International Conference. Conference on Intelligent Computer Mathematics*, Calculemus '09/MKM '09, pages 122–137, Berlin, Heidelberg, 2009. Springer-Verlag.
38. G. O. Passmore and P. B. Jackson. Abstract Partial Cylindrical Algebraic Decomposition I: The Lifting Phase. In S. B. Cooper, A. Dawar, and B. Loewe,

editors, *Proceedings of Computability in Europe 2012: Turing Centenary (To appear)*. Springer-Verlag, 2012.

39. L. Paulson. *Logic and Computation: Interactive Proof with Cambdrige LCF*, volume 2. Cambridge University Press, 1987.
40. L. Paulson. Isabelle: The next 700 theorem provers. In *Logic and Computer Science*, pages 361–386. Academic Press, 1990.
41. D. A. Plaisted. The Search Efficiency of Theorem Proving Strategies. In *Proceedings of the 12th International Conference on Automated Deduction*, CADE-12, pages 57–71, London, UK, UK, 1994. Springer-Verlag.
42. D. Prawitz. An Improved Proof Procedure. *Theoria*, 26(2):102–139, 1960.
43. A. Robinson. Short Lecture. Summer Institute for Symbolic Logic, Cornell University, 1957.
44. J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12:23–41, January 1965.
45. H. Wang. Toward mechanical mathematics. *IBM J. Res. Dev.*, 4:2–22, January 1960.
46. V. Weispfenning. Quantifier Elimination for Real Algebra - the Quadratic Case and Beyond. *Appl. Algebra Eng. Commun. Comput.*, 8(2):85–101, 1997.
47. L. Wos, D. Carson, and G. Robinson. The unit preference strategy in theorem proving. In *Proceedings of the October 27-29, 1964, fall joint computer conference, part I*, AFIPS '64 (Fall, part I), pages 615–621, New York, NY, USA, 1964. ACM.
48. L. Wos, G. A. Robinson, and D. F. Carson. Efficiency and completeness of the set of support strategy in theorem proving. *J. ACM*, 12:536–541, October 1965.
49. H. Zankl and A. Middeldorp. Satisfiability of non-linear (ir)rational arithmetic. In *Proceedings of the 16th International Conference on Logic for Programming and Automated Reasoning*, volume 6355 of *Lecture Notes in Artificial Intelligence*, pages 481–500, Dakar, 2010. Springer-Verlag.