# User Control and Direction of a More Efficient Simplifier in ACL2

Rob Sumners
Advanced Micro Devices, Inc.
Austin, Texas, USA
robert.sumners@amd.com

## ABSTRACT

We present an efficient term simplifier written in ACL2 and interfaced with ACL2 as an untrusted clause processor. We also demonstrate how an advanced user can extend this simplifier in a sound manner by proving rewrite rules with special annotations and programmed constraints on their application. For problems requiring extensive case analysis, the simplifier is more efficient than ACL2 built-in simplification and we demonstrate this on some relevant examples. In addition, we discuss the issue of user control over predictable simplification and conclude the paper with the proposed implementation of invariant discovery using the simplifier.

## Categories and Subject Descriptors

D.2.4 [**Software Verification**]: Formal Methods

## General Terms

Verification

## Keywords

Theorem Proving, Term Rewriting, Clause Processor

## 1. MOTIVATION

In the course of proving theorems in ACL2, the vast majority of the computational resources are spent in the simplification stage of the prover. This is not too surprising when one considers that the majority of the ACL2 theorem proving code implementing the "waterfall" is dedicated to the simplification stage. It is, therefore, relevant to consider alternatives or extensions to ACL2 simplification – especially in cases in which it takes considerable human and computational resources to get the ACL2 simplifier to produce the desired result.

In addition, for many problem domains tackled in ACL2 (*e.g.* proofs of concurrent program correctness), theorems are either proven directly by simplification or proven by a single induction followed by simplification. As an example, all of the proofs for the correctness of a concurrent deque in [13] in ACL2 were either proven by simplification or by a single induction followed by simplification. This highlights the importance of efficient simplification in the proofs of theorems about systems and program execution in ACL2.

An alternative to using ACL2 for brute-force simplification is to use ACL2 to reduce the theorem to be proven to a problem that can be translated into a decidable logic that can be solved by an external tool (usually a propositional SAT checker)[4]. This approach is attractive because it separates the problem domains of the tools involved. The user may write high-level definitions in the more expressive language of ACL2 and use the theorem proving support in ACL2 to manually transform the problem to a tractable domain, which is handled by a decision procedure that can be written efficiently to handle problems in the more tractable domain. In cases where the manual effort required to transform the problem to the tractable domain is manageable, this process can certainly be effective, but in cases where this translation is difficult (*i.e.* when dealing with a concurrent system with an unbounded number of processes), then the translation may require considerable work on proofs and invariant definitions. A further issue arises when either the translation process or the decision procedure is ineffective in handling the problem efficiently – the user often only has coarse-grain control or guidance in the process.

In other theorem provers, variations of Nelson-Oppen[7] or Shostak's [11, 8, 10] are often used as a method to integrate efficient decision procedures into a more general proof engine. ACL2 also integrates decision procedures into the simplification process. In each of these cases, the integration of decision procedures requires careful translation of results between procedures to avoid soundness and completeness issues. The proposal of this paper is to implement decision procedures as sets of rewrite rules with the common term representation as the translation between decision procedures with no inherent soundness issues or translation issues between procedures.

The approach we present in this paper can be viewed as an alternative or extension to ACL2 simplification. We present an efficient simplifier based on term rewriting using rewrite rules derived from theorems proven by the ACL2 user – similar to the use of rewrite rules in ACL2 simplification. This new efficient simplifier is written in ACL2 and is used as a trusted clause processor in certified books. The simplifier

is essentially an optimized term rewriter with support for fine-grained control provided to the user through annotated rewrite rules. Through these annotated rewrite rules, the user can control the simplifier and actually implement local procedures for simplifying certain types of operations and logical relationships. The control that the user is given is principled in that the rewrite rules are derived from proven theorems and, in theory, cannot break the soundness of the results of the simplifier.

In support of the control afforded to the user, the simplifier provides feedback on its execution to the user on several levels. The simplifier always provides statistics on the number of nodes created, the types of nodes created, and various rewrite rule applications and attempts. When a theorem fails, a concise report is generated on the failing case. Further, the simplifier itself is designed to be straightforward and predictable to allow the user to have a more direct impact on its behavior through rewrite rules.

In this paper, we will present the architecture of this simplifier and some example applications. The definition of the simplifier is provided in the supporting materials and is a derivative of the work presented in [14] (which includes an informal argument for soundness and relative completeness).

## 1.1 Supporting Materials

The supporting materials for this paper consist of two files: `kas.lisp` and `examples.lisp`. The `examples.lisp` file shows some applications of KAS on a pipeline example which is detailed in Section 4.1.

The file `kas.lisp` contains the code defining the simplifier. The file is essentially divided into three parts. The first part of the file is the definition of several macros that are used to make the main definition section of the simplifier easier to read and manage but also introduce various type declarations for efficient execution. The second part of the file is the definition of the simplifier; the main mutual recursive function clique is entered through the `rewrite-node` function. This simplifier definition will be the focus of the next section. The final part of `kas.lisp` is a set of functions that interface the simplifier to the ACL2 world and state. This code extracts rewrite rules, function definitions and properties, and various other pieces of information from the ACL2 world.

We highly encourage the interested reader to look over these files, load them in ACL2, and play around with the simplifier. The primary example problem provided is a good candidate for testing and analyzing the operation of the simplifier. In addition, one of the benefits that we claim about this simplifier is that it is written in ACL2 and in a manner that affords examination, understanding, and even extension by an ACL2 user. In fact, a mechanical proof of the soundness of the simplifier in ACL2 is an ongoing effort.

It is important that we note that the simplifier definition requires a 64-bit LISP compiler and runtime. In particular, the code has been rewritten to take advantage of the larger fixnums on 64-bit systems. All results in this paper were achieved running OpenMCL on MacBook with a 2GHz processor with 1GB RAM. It is recommended that users run the simplifier with Clozure CL on the platform of their choice. Running the simplifier with a Common Lisp that does not support larger fixnums will likely result in poor performance.

# 2. SIMPLIFIER ARCHITECTURE

The ACL2 simplifier is written in the ACL2 programming language in an applicative style.[1] The applicative style reduces the inherent complexity of the functions defining the simplifier but at the cost of efficiency. Further, the ACL2 simplifier includes several procedures and heuristics to improve the effectiveness and efficiency of the simplifier, but with the downside of increasing the complexity of the simplifier. To improve control of the prover, most of these procedures and heuristics are extended with mechanisms to support user-specified directives. Our proposal is to define a simplifier that is more efficient and less complex and affords greater control than the ACL2 simplification stage of the ACL2 prover. Our intent is to make the simplifier as effective at proving theorems as the ACL2 simplifier, but not at the cost of the other goals. This simplifier is named KAS which stands for Kernel Architecture Simplifier. This name refers to an architecture based on a principle of isolation and separation analogous in many ways to the kernel architectures of "modern" operating systems.

KAS is best described as an elaborated, optimized, inside-out, ordered, conditional term rewriter with support for fine-grained user control. In general, a simple term rewriter would not be considered a simplifier since its efficiency and effectiveness would be insufficient for larger theorems requiring analysis of a large number of cases. Part of the proposal for KAS is an argument for implementing decision procedures and heuristics within the basic language or structure of extended term rewriting. For this proposal to be reasonable, the term rewriter needs to provide sufficient optimization to approximate the efficiency of decision procedures applied to the problems that commonly arise in the application of ACL2. The definition of a simple inside-out, ordered term rewriter provided in Figure 1 – this simple rewriter is a toy definition and will merely provide a point of reference in the presentation of KAS.

The function `simple-rewrite` is the entry point to the simple rewriter in Figure 1 and takes a term `trm` and returns a term resulting from the rewriting of `trm` using proven theorems as rewrite rules. The function `rewrite-term` takes a term `trm` and a list `ctx` of equations that hold in the current context for rewriting. This context `ctx` will be extended when the true branch or false branch of an `if` term is rewritten and, in this manner, `if` terms define the pivot points for further case splitting. The function `rewrite-term` returns the fixpoint in the repeated application of the single-step rewrite function `rewrite-step`.

The function `rewrite-step` takes a term and a context and splits into different cases. If the current context is inconsistent or the term is a quoted constant, then the input term is simply returned. Otherwise, if the input term is equated in

[1]The ACL2 simplifier does not extensively utilize single-threaded objects or arrays, and generally uses constructive lists for most of the operations it performs. The simplifier does utilize Common Lisp symbol property lists to provide fast lookup of properties attached to functions.

```
(defun variablep (x) (not (consp x)))
(defun constantp (x) (and (consp x) (eq (first x) 'quote)))
(defun equationp (x) (and (consp x) (eq (first x) 'equal)))
(defun true      (x) (and (constantp x) (second x)))
(defun false     (x) (and (constantp x) (not (second x))))
(defun not!      (x) (list 'equal x ''nil))
(defun get= (x a)
  (cond ((endp a) nil)
        ((equal x (second (first a))) (first a))
        (t (get= x (rest a)))))
(defun add= (trm ctx) (cons trm ctx))
(defun assume= (trm ctx)
  (cond ((false trm)     'inconsistent)
        ((equationp trm) (add= trm ctx))
        (t               ctx)))

(mutual-recursion
(defun apply-rule (trm rl ctx)
  (let ((bnd (unify (rule-lhs rl) trm)))
    (if (and (unify-was-success bnd)
             (true (rewrite-term (subst (rule-hyps rl) bnd) ctx)))
        (subst (rule-rhs rl) bnd)
      trm)))

(defun try-rules (trm rls ctx)
  (if (endp rls) trm
    (let ((trm+ (apply-rule trm (first rls) ctx)))
      (if (equal trm+ trm) (try-rules trm (rest rls) ctx) trm+))))

(defun rewrite-if (args ctx)
  (let ((tst (rewrite-term (first args) ctx)))
    (list tst
          (rewrite-term (second args) (assume= tst ctx))
          (rewrite-term (third args) (assume= (not! tst) ctx)))))

(defun rewrite-list (lst ctx)
  (if (endp lst) () (cons (rewrite-term (first lst) ctx)
                          (rewrite-list (rest lst) ctx))))

(defun rewrite-args (args fn ctx)
  (cond ((eq fn 'hide) args)
        ((eq fn 'if)   (rewrite-if args ctx))
        (t             (rewrite-list args ctx))))

(defun rewrite-step (trm ctx)
  (cond
   ((or (inconsistent ctx) (constantp trm)) trm)
   ((get= trm ctx)
    (rewrite-term (third (get= trm ctx)) ctx))
   ((variablep trm) trm)
   (t (try-rules (cons (first trm)
                       (rewrite-args (rest trm) (first trm) ctx))
                 (get-rules (first trm)) ctx))))

(defun rewrite-term (trm ctx)
  (let ((trm+ (rewrite-step trm ctx)))
    (if (equal trm+ trm) trm (rewrite-term trm+ ctx))))
)
(defun simple-rewrite (trm) (rewrite-term trm ()))
```

**Figure 1: Simple Inside-Out Ordered Term Rewriter**

the current context, then the result of rewriting the term it is matched with in the current context is returned. Otherwise, if the input term is a variable symbol, then it is simply returned. If none of these cases apply, we rewrite the term by first rewriting the arguments of the term and then attempting to apply rewrite rules. Rewriting the arguments of the term consists simply of rewriting the arguments in sequence, unless the function symbol is `if`. In this case, the true-branch and false-branch terms are rewritten with appropriately extended contexts.

The function `try-rules` goes through each of the rewrite rules attached to a given function symbol in order until a rule is found that can be applied. A rule can be applied when its left-hand-side `lhs` can be equated with the input term under a binding `bnd` of the variables in the `lhs` and where the hypothesis `hyps` of the rule rewrites to truth in the current context after a substitution using the unifying binding `bnd`. If a rule can be applied, then the right-hand side `rhs` of the rule is returned after a substitution using the unifying binding `bnd`.

The function `simple-rewrite` defines a simple, ordered, inside-out rewriter that applies conditional rewrite rules. The ACL2 term rewriter component of the ACL2 simplifier bears some resemblance to this simple term rewriter in the sense that it operates on ACL2 objects representing terms and primarily supports ordered, inside-out, conditional rewriting. The ACL2 term rewriter also maintains a structure named the `type-alist` that associates terms with information known about the terms and this structure is updated when rewriting the true and false branches of `if` terms.

This simple term rewriter also resembles the KAS procedure we present in this paper. Indeed, KAS operates recursively on term structures in much the same manner as this simple rewriter but with several optimizations. The optimizations in KAS operation fall into three categories: memory management, memoization and context management, and miscellaneous optimizations and features. We consider each group of optimizations in turn and then describe the mechanisms provided in KAS for user control and interfacing with ACL2. It is important to point out that each optimization was carefully considered before its addition to KAS. Indeed, several optimizations were considered and even implemented only to be removed after analysis demonstrated that the benefit in performance did not outweigh the cost in complexity. Optimizations can significantly complicate function definitions and one of the primary objectives for KAS is to reduce complexity to facilitate future maintenance and extensions, clear predictable operation and control, and, eventually, a mechanical proof of soundness.

## 2.1 Terms and Memory Management

The simple rewriter in Figure 1 represents terms in their syntactic form as ACL2 objects. This representation affords elegant functions for manipulating terms, but is inefficient in many ways. First, this representation is not compact. Second, this representation does not afford the direct "tagging" of terms with information computed about the term. Finally, this representation may not provide for a sufficient amount of structure sharing among subterms. We address

these issues with this first group of optimizations focusing on representations of terms and memory management in KAS.

**Managing Representations of Terms.** The use of `cons` to build lists representing function calls does not afford a compact representation of a term. In most Common Lisp implementations, a `cons` consists of three words of memory. The first word stores a header field that tags the three words in memory as a `cons` structure (along with some auxiliary information such as traversal bits used during garbage collection), while the other two words in memory define references or pointers to the `car` and `cdr`. This description of a `cons` structure is a generalization and Common Lisp compilers do differ, but – as of this writing – all Common Lisp compilers require at least two memory words to store the references to the `car` and `cdr` objects. So, for example, an `if` function call in a term will require 4 `cons` structures and from 8 to 12 words of memory. Furthermore, the `cons` structures defining a function call need not be allocated "near" each other in memory, which will limit the usefulness of caching structures in modern microarchitectures that leverage this locality in memory. Further, it is important to make the representation of a function call as compact as possible to fit more function calls into a single cache line.

Our direct approach is to represent every function call in a term as consecutive words of memory forming a *node*. This is achieved in ACL2 by creating an array of fixnums `node-arr` in a stobj and defining a node as a natural number index into this array (similar to a pointer to a struct in C). The fields of a node `x` are then defined by the fixnums at index `x`, `(+ x 1)`, `(+ x 2)`, ... `(+ x k -1)`. The first fixnum at index `x` identifies the function operator and thus the arity of the node. The size of the node `k` then includes fixnums needed to store the header (some fixed number of indexes depending on the information stored with the node) and a fixnum storing a node for each argument in the function call. The next node stored in the array will start at the index `(+ x k)` and so on. This representation reduces by half (at least, and usually more) the amount of storage required in comparison to `cons` structures in current Common Lisp implementations, and the indexes for the arguments of a node are close in memory to the node itself. Every node `x` that is constructed defines a term in the ACL2 logic `(node-to-term x)`.

While this node representation is more compact, there are a few downsides that need to be addressed. First, the representation is not as elegant as `cons` structures. This is simply true, but we can alleviate this downside with the use of macros to hide the details of the underlying structure. Second, the stobj containing the array field that stores the nodes will need to be passed to all functions that access or update nodes. As we will see later, the use of a stobj was already required to afford fast access and destructive updates to arrays within the applicative semantics of the ACL2 programming language, and macros can again alleviate much of the syntactic burden of using stobjs. The final potential downside to the representation of terms as nodes in KAS is that KAS will need to perform its own garbage collection of memory. In the case of KAS, an efficient and elegant scheme for managing the allocation of nodes is utilized, which will be more efficient than a generic Common Lisp garbage collection process operating on terms stored as `cons` structures. Thus, this downside will actually prove to be a positive of the KAS approach since the Lisp garbage collector will have far fewer Lisp objects to manage for collection.

**Unique Construction of Nodes.** During rewriting, the same term may be created in several different rewrites and as a subterm in several different terms. In most cases, these different instances of the same term will not be the same object in memory, so there is opportunity for structure sharing by creating a node only once – any time a new term is needed, it can be looked up in a hashtable that associates a function symbol and its arguments with a uniquely constructed node representing this term. This is particularly useful in contexts in which structure sharing has demonstrated exponential improvements in space requirements [2, 1]. The hashtable requires (on average) an additional two words of memory per node, and some execution time cost is incurred during the creation of new nodes to determine if the new node already exists. An additional benefit of unique node construction is that testing whether two nodes are equivalent reduces to the equality of the two fixnum indexes (often compiled into a single instruction testing machine word equality). Further, unique node construction will afford the direct lookup of any memoized information stored with the node rather than looking into a separate structure to see if any memoized information has been stored for an equivalent node. Even with these benefits of unique construction, the costs of unique node construction warrant some consideration. In KAS, only a subset of the nodes that are created will be uniquely constructed. This subset of nodes will be termed the *promoted* nodes, and the determination of which nodes are promoted has effects on several aspects of KAS operation.

**Node Allocation and Promotion.** One downside of having KAS manage the memory allocation for nodes is the need to reclaim unused nodes. One strategy would be to simply never reclaim nodes. This simple strategy might be reasonable if the need to reclaim nodes were sufficiently mitigated by the structure sharing achieved with unique node construction. In practice, this strategy is ineffective due to the large number of "junk" nodes that are created during rewriting in KAS operation. This is especially true given the philosophy of KAS to use rewriting as the basis for simplification with rewrite rules performing node transformations as the steps of some algorithmic procedure. As an example of "junk" node creation, non-recursive function definitions will introduce unconditional rewrite rules that will always fire unless the user explicitly disables the rules. In many cases, these rules will be enabled and, thus, any node whose outermost function symbol is a non-recursive function symbol will be immediately rewritten to the node resulting from the expansion of the non-recursive function. Another common source of "junk" nodes arise from normalizations of nodes that occur through a sequence of rewrites. For example, a common process for normalizing `if` structures is by so-called "if-lifting", which transforms any term into a term where the only `if` subterms are in the true and false branches of other `if` terms. In KAS, if-lifting would be implemented using rewrite rules such as the following:
`(equal (if (if x y z) a b) (if x (if y a b) (if z a b)))`.
If-lifting rules will create numerous intermediate nodes that will immediately be rewritten into a different node and which

will likely never be created in any other rule application. Due to these properties, it would be beneficial to reclaim these intermediate nodes immediately and efficiently after their construction.

The mechanism we use for managing node allocation and reclamation is termed *promotion*. All nodes are initially allocated as "junk" or *transient* nodes. Certain transient nodes may then be promoted while the others will be reclaimed. A node is promoted if (a) the node represents a quoted constant or a variable symbol, (b) the node cannot be rewritten (*i.e.* it is in *normal-form*) in the current context, or (c) the node's arguments are promoted and an equivalent transient node was previously constructed. An invariant on all promoted nodes is that their arguments must also be promoted. Promoted nodes are constructed uniquely and are never reclaimed. Further, promoted nodes have additional fields stored with them that memoize certain computations. Transient nodes require minimal storage per node and are not stored uniquely. Reclamation of transient nodes is efficient and simple: for certain functions that are known to return promoted nodes (*i.e.* the function in KAS which correlates to `rewrite-term` in Figure 1), the current transient node allocation index is saved on function entry and then restored immediately before returning from the function. In this manner, any transient nodes that were allocated during the execution of the function are immediately reclaimed on the return from the function. For most common applications of KAS, the vast majority of the nodes that are constructed will be transient and this node promotion strategy will keep the memory requirements of KAS tractable even for larger problems. Further, in situations in which the key properties of promoted nodes (unique construction, memoization, etc.) were likely to benefit execution time are situations in which we strongly expect the nodes to satisfy one of the conditions (a), (b), or eventually (c) necessary for promotion.

## 2.2 Memoization and Context Management

To avoid repeating numerous previous (possibly long) sequences of rewrites, it is important to store results of rewrite sequences efficiently and recall these results when needed. It is also important to determine efficiently whether a node is equated with another node in the current context (as in the call of `get=` in the simple rewriter). These two needs can be addressed with the same mechanism we call a representative or *repnode*. A *repnode* is an index stored in the header of a promoted node. It is an invariant that at all times during execution, a node is equivalent to its repnode assuming the current context. We describe how repnodes are used to store contextual information and memoize results of previous rewrites.

**Storing and Accessing Contexts.** When the simple rewriter in Figure 1 rewrites the true and false branches of an `if` term, it extends the current context `ctx` using the function `assume=`, which takes the current context `ctx` and extends it by assuming the current rewritten test argument `tst` of an `if` operator to either be true or false. In the function `rewrite-step`, the current context is consulted using the `get=` function to determine if the term is known to be equal to a different term; if so, this term is returned. The call of `get=` requires a linear search through the `ctx` to determine if there is a match to the current term. In KAS, the extension of the current context performed by the function `(add= x y c)` would consist simply of setting the repnode for the node `x` to point to the node for `y`. When the KAS rewriter reaches a node that has a repnode, then the rewrite of the repnode is returned.

**Memoizing Rewrites in Contexts.** During the simplification of a theorem, a large number of rewrites are performed and many of the rewrites are repetitions of previous rewrites. It is thus useful to store and retrieve records or memos of previous rewrites efficiently. In KAS, this memoization is stored in the repnode field of a given node, which provides fast constant-time lookup for memo results.

The repnode is stored in the header of a promoted node. An invariant of KAS execution is that the repnode $r$ of a promoted node $x$ is equivalent under the current context. The *context* is a stack of assumed equalities under which terms are rewritten. Similar to the simple rewriter in Figure 1, when the true or false branch of an `if` term is rewritten, then the context is extended with an equality from assuming the test of the `if` term is either true or false. A straightforward approach would be to store any equalities derived from rewriting in the repnode fields of promoted nodes and then revert these repnode equations when an equality is popped from the current context. The problem with this naive approach is that many equalities derived from rewrites are either independent of the context or only dependent on part of the current context.

In KAS operation, all computed repnode equalities are tagged with the subset of the assumptions from the current context that are sufficient for the computed equality to be valid. For example, when an `if`-node test is assumed true or false, then it is tagged with the singleton set with the current depth of the context stack. These subsets of the context are stored as bitvectors in the header of a node (along with the repnode) where each bit position in the bitvector corresponds to a position in the current context stack. Further, these context subset bitvectors are returned with every function in the main mutual recursive function clique in KAS which corresponds to the functions in Figure 1. Each of these rewrite functions will take a node and return the node resulting from rewriting and a bitvector encoding the assumptions from the current context which were used in the rewriting. When a repnode is stored for a promoted node, the bitvector justifying the equation is also stored in the header of the promoted node. The bitvector for the composition of rewrite steps is the union (computed as `logior`) of the bitvectors for each step. When rewriting an `if` node, the resulting bitvector includes the union of the bitvectors for the rewrite of the true and false branches but drops the bit position in the context used for the assumption of the test as true or false. When the current context is popped, KAS will only revert the repnode equalities, which are dependent on the assumption that is popped off the context.

A simple example of rewrites that are independent of context are computed facts about the types of terms. Many conditional ACL2 rewrite rules will include tests of whether a certain term is of a certain type (*e.g.* `true-listp`) in their hypothesis; in many cases, these tests are either true or false properties of the outermost function in the term. In

ACL2 simplification, a `type-alist` is used to store typing information computed about terms in the current clauses with assistance provided by the user in the form of `type-prescription` and `forward-chaining` rules. In KAS, there is no analogue to the type-alist; instead, it is expected that terms representing type information (*e.g.* `(type-alist (foo x))`) will rewrite to `T` independent of the current context. In KAS, this rewritten equality will be tagged as independent of the assumptions in the context, which will cause the equality to persist through the operation of KAS unaffected by updates to the context. These operations on repnodes and context bitvectors are implemented efficiently in KAS, which is critical since these repnode equations are the primary mechanism in KAS for memoizing previous computations.

## 2.3 Additional Optimizations in KAS

We briefly highlight some of the additional optimizations and efficiency concerns in the implementation of KAS. There are several optimizations in KAS we do not detail in this paper, but the discussion here will provide an idea of the types of efficiency issues that have been addressed.

**Avoiding Overhead of Lisp Execution.** Execution of functions written in applicative Common Lisp incurs certain costs due to the nature of applicative Lisp evaluation. Compared to programs written in C, programs in applicative Lisp require more space for less compact structures such as lists instead of arrays and boxed integers instead of machine words. Programs in applicative Lisp also require more execution time due to inefficient data structures, overhead for function calls, and overhead for garbage collection. Each of these costs is minimized in the definition of KAS. Stobjs with arrays are used to avoid the costs associated with allocating, accessing, and managing lists. Careful type declarations and limits on integer variables avoid the use of boxed integers and allow for efficient operations directly on machine words encoding integers, bitvectors, etc. Function calls are removed through the use of macros to inline smaller non-recursive functions and the writing of recursive functions to be tail-recursive when feasible. The KAS main loop only uses arrays, integers, and booleans (*i.e.* no `cons`ing), which not only ensures the use of efficient data structures but avoids the creation of garbage that would otherwise require collection.

**Specialized Data Structures.** Several specialized data structures are utilized in the implementation of KAS. These structures are optimized to match the frequency of certain necessary operations performed on the structures. As an example, we consider the implementation of the *undo-stack*, a key component of the implementation of contextual memoization in KAS. In KAS, a single large stobj `ls$` called the *logic state* is passed around all of the functions defining KAS. The logic state includes the storage of the nodes along with all of the data structures needed to implement KAS operation including the undo-stack. When the context is extended in order to rewrite the true and false branches of an `if` node, the `ls$` stobj is destructively updated to include the assumption of the equality defined by the test node. Further, whenever a rewrite is memoized, the repnode of a node is destructively updated. After rewriting the true or false branch of an `if` node, any repnode updates associated with the assumption of this test as true or false must be rescinded, and this is where the undo-stack is utilized. The undo-stack is structured as a separate stack of entries for each integral position in the current context. When a repnode is updated, the context dependence set is examined to determine which stack in the undo-stack is used to store the entry marking the repnode update. When the context is reverted, the entries in the undo-stack corresponding to the assumption of the test node will be undone to restore the logic state to a consistent state before the assumption of the context. The design of the undo-stack affords constant-time updating and reverting of repnode updates in KAS while maintaining a logic state consistent with the current context.

**Avoiding Repeated Computation.** We presented the use of the repnodes in KAS as a mechanism to memoize previous rewrite results. The KAS implementation supports additional mechanisms to avoid repeating computations. As an example, after rewriting the arguments for a node, if the arguments have not changed and the node has been rewritten to normal form previously, and the only rewrite rules that might be applied are unconditional, then the original node is returned immediately since no rewrite rules will match the node. This mechanism does not effect the relative completeness of KAS since the rewrite rules which would have been attempted would not have matched. The user can also cause this optimization to be enabled for functions with conditional rewrite rules, but this may effect completeness since the procedure may not attempt a rewrite that would have succeeded in the current context. Due to this optimization, it is beneficial for users to try to avoid conditional rewrite rules for operators that will occur frequently in proofs.

## 2.4 User Control and Interfacing with KAS

The user of ACL2 generally proves theorems of the form `(implies` $\alpha$ `(equal` $\beta$ $\gamma$`))` and these theorems are then used as conditional rewrite rules in subsequent proof efforts. ACL2 provides extensions of the basic treatment of a theorem as a conditional rewrite rule through the use of various forms of tagging. For instance, the user might tag the theorem with a different rule class and cause the theorem to be used for type prescription or forward chaining. The ACL2 user can also use the special operators `syntaxp` and `bind-free` to attach a user-defined function call to a rewrite rule, which can provide an additional test on whether the rewrite rule should fire beyond the unification of the `lhs` and relieving of the `hyps`. The ACL2 user can also control the current set of enabled rewrite rules used to prove a theorem (or a specific subgoal) by enabling and disabling certain rewrite rules. The control afforded an ACL2 user is varied and powerful, but it is also complex and somewhat incomplete (*e.g.* the user cannot enable or disable a rewrite rule as a side effect of applying another rewrite rule).

Similar to ACL2, KAS uses previously proven theorems as conditional rewrite rules, but KAS only supports conditional rewrite rules. The other rule classes supported by ACL2 (*e.g.* type prescription) have no counterpart in KAS. KAS does not directly support the mechanisms provided in ACL2 to control the current enabled rewrite rules or the extended

| operation | side effect |
|---|---|
| set-var-bound | bind a free variable in a rewrite to a node |
| set-rule-sieves | modify the current filters attached to a rule |
| set-rule-enabled | enable or disable a rewrite rule |
| set-rule-ctr | modify counter for number of rule applications |
| set-node-step | set node allocation incremental step |
| set-node-limit | set node allocation limit (junk and promoted) |
| change-rule-order | change the order of rewrite rules |
| set-rule-traced | enable or disable trace output of a rule |
| set-user-mark | set or clear a boolean mark on a node |

**Figure 2: Permitted Side Effects from Filter attached to Rewrite Rule**

application of certain rewrite rules. KAS does support the special `hide` operator, which effectively disables the rewriter on a given term.

Instead of the variety of control mechanisms that ACL2 supports, KAS only supports fine-grained user control through the tagging of certain conditional rewrite rules with *filters* introduced with the `sieve` operator. A sieve can be seen as an extension of the `syntaxp` and `bind-free` special operators in ACL2. In ACL2, the `syntaxp` operator allows the user to attach the non-`NIL` evaluation of a user-defined predicate as an additional requirement on the application of a rewrite rule. The `sieve` operator in KAS also allows the user to attach a user-defined predicate (or filter) to a rewrite rule but extends this `syntaxp` principle in several facets. First, a filter function can efficiently query the current logical state stobj to determine which nodes are equated, how many times a rule has fired, etc. Second, a filter function can use a stobj to implement efficient data structures (*e.g.* hash-tables) and to store information for subsequent calls of any filter function. Further, filter functions can return a list of commands that can be used to modify the logical state stobj – but the modifications of the logical state are restricted to ensure the logical soundness of KAS. The possible modifications of the logical state are restricted to the operations listed in Figure 2. We will demonstrate the use of a key filter function for case splitting in Section 4.

# 3. EXAMPLE 1: BDDS

The sole propositional logic operator in KAS is `if` and, generally, the most common operator arising in nodes created and manipulated during KAS operation is the `if` operator. The proper management of `if` nodes through rewrite rules and sieves is essential for the effective usage of KAS. A straightforward approach is to use if-lifting where a term is rewritten to a form where all `if` tests have no `if` subterms and no functions other than `if` have an `if` subterm. In this form, all tests are rewritten to normal form and all branches of the `if` are rewritten to normal form with complete context information.

If-lifting is straightforward and may be sufficient, but is often far too inefficient for use in theorems with a large number of cases. In some situations, the problem with simple if-lifting is that it does not normalize the structure of the `if` terms sufficiently to afford the potential savings achieved in structure sharing and memoization. For propositional terms with more regular structure, the potential benefit from structure sharing can be significant, and often the best option in these

```
(defun bv (x) (if x t nil))

(defun bits-equiv (x y)
  (if (endp x) (endp y)
    (and (consp y) (iff (car x) (car y))
         (bits-equiv (cdr x) (cdr y)))))

(defthm bits-equiv-nil-reduce
  (equal (bits-equiv () ()) t))

(defthm bits-equiv-cons-reduce
  (equal (bits-equiv (cons a x) (cons b y))
         (and (iff a b) (bits-equiv x y))))

(defthm bits-equiv-symmetric-5
  (let ((x (list (bv x4) (bv x3) (bv x2) (bv x1) (bv x0)))
        (y (list (bv y4) (bv y3) (bv y2) (bv y1) (bv y0))))
    (equal (bits-equiv x y) (bits-equiv y x))))
```

**Figure 3: BDD Example Application**

```
(defthm bdd-if-lift-test
  (equal (if (if x y z) m n) (if x (if y m n) (if z m n))))

(defthm bdd-bv-then-split
  (equal (if x (bv y) z) (if x (if (bv y) t nil) z)))

(defthm bdd-bv-else-split
  (equal (if x y (bv z)) (if x y (if (bv z) t nil))))

(defthm bdd-reorder-then
  (implies (sieve (bdd-order a x))
           (equal (if x (if a b c) y)
                  (if a (if x b y) (if x c y)))))

(defthm bdd-reorder-else
  (implies (and (sieve (bdd-order a x)) (sieve (bdd-order a y)))
           (equal (if x y (if a b c))
                  (if a (if x y b) (if x y c)))))
```

**Figure 4: Binary Decision Diagram Rewrite Rules**

cases is the use of so-called Reduced Ordered Binary Decision Diagrams[2] or BDDs. BDDs are a normal-form representation of propositional formulae consisting of `if` nodes in which the sequence of propositional variables that are reached along any path from the top node to a terminal node is consistent with a total order on all propositional variables. With the appropriate ordering on variables and utilizing structure sharing, BDDs afford compact representations of many common operations on bitvectors encountered in hardware description and low-level software. For example, consider the theorem `bits-equiv-symmetric-5` in Figure 3. With the variable order `x4`,`x3`,...,`y4`,`y3`,..., the BDD generated from the term `(bits-equiv x y)` will afford no structure sharing and lead to a BDD roughly of size $2^5$. With the variable order `x4`,`y4`,...,`x0`,`y0`, the generated BDD does support structure sharing and is roughly of size $2 * 5$.

The rewrite rules in Figure 4 will normalize an `if` term to satisfy the BDD variable ordering property. The particular order is defined by the filter function `bdd-order` and a default ordering scheme is provided that can be modified by the user to target the term structure of a specific theorem. The rewrite rules from Figure 4 are attempted in reverse order, which allows us to remove the second filter of `(bdd-order a y)` in the theorem `bdd-reorder-then`.

# 4. EXAMPLE 2: CASE SPLITTING

For theorems requiring extensive case analysis, the unrestricted use of if-lifting, BDD, or other normalizing rules would lead to excessive and inefficient rewriting of `if` nodes. Unless the target theorem has certain structure that the user can leverage, the blind normalization of `if` structures will not scale effectively to larger and more complex theorems. Most theorems that arise will require efficient case analysis without normalizing the propositional structure of the terms created. This is one of the primary benefits of propositional satisfiability checkers implementing variants of the Davis-Putnam procedure[15, 6, 12]. These checkers operate on unnormalized representations of propositional formulae by iteratively splitting on propositional variables and reducing the resulting formula under the assumption of the propositional variable as either true or false. Satisfiability checkers have become the most robust and efficient means to solve propositional theorems and problems that can be encoded into propositional theorems[3].

This approach has the significant additional benefit of potentially returning sooner on failed proof attempts. This early failure property is far more important in the context of ACL2 theorems for two reasons. First, most of the ACL2 theorems that require significant case analysis will also require significant user interaction; any delay in reporting failed proofs will be amplified considerably in the time the user requires to complete the proof. Second, as the definitions and lemmas for a complex theorem are adjusted to fix previous failures, the speed of simplification will often increase considerably. The reason for this speedup is that many of the cases which that previously led to failures will now resolve to truths much sooner in the rewriting process. This avoids unnecessary and costly expansion of unnecessary nodes and case analysis due to `if` subterms.

We support an efficient case-splitting heuristic through the use of rewrite rules tagged with filters. The primary rewrite rules supporting case splitting are provided in Figure 5. The rewrite rules operate as follows. The user triggers case splitting by wrapping the target term $\alpha$ with the `prv` operator to create `(prv` $\alpha$`)`. When KAS attempts to rewrite the term `(prv` $\alpha$`)`, it will first rewrite $\alpha$ to a normal form and will then attempt (due to reverse chronological rule ordering) the use the rule `prv-case-split`. The `prv-case-split` rule calls the filter function `case-split`, which determines if there are any candidate nodes to choose for case splitting. If such a candidate node is found, then the free variable `C` in the rewrite rules is bound to this candidate node and the rewrite rule is applied. Otherwise, the rewrite is not applied and `(prv` $\alpha$`)` rewrites to $\alpha$ by the application of the `prv-evaporates` rule. If `prv-case-split` was applied, then we have a node of the form `(prv2 (if` $\beta$ `(prv3` $\alpha$`) (hide` $\alpha$`)))`. Due to inside-out rewriting, `(prv3` $\alpha$`)` is rewritten first under the assumption of $\beta$. If assuming $\beta$, $\alpha$ reduces to `T`, then `(prv3` $\alpha$`)` will reduce to `T` via the rule `prv3-drop-if-T`. In this case, the rule `prv2-then-pass-shift` will apply to the `prv2` node and replace the `hide` around the else branch with a `prv`. Otherwise, with the assumption of $\beta$, $\alpha$ did not reduce to `T`, and the rule `prv2-if-gen-prv` is applied to replace the `prv3` on the then branch with a `prv` to enable further case-splitting in the then branch. If neither `prv2-if-gen-prv` nor `prv2-then-pass-shift` can be applied, then the `prv2-evaporates` is applied to remove the `prv2` operator. Through the application of these rules, a term is rewritten by case-splitting into an `if` tree whose leaves are either `T`, an application of `hide`, or the current node, which is being targeted through `prv`.

```
(defun prv (x) x) (defun prv2 (x) x) (defun prv3 (x) x)

(defthm prv3-drop-if-t (equal (prv3 t) t))

(defthm prv2-evaporates
  (equal (prv2 (if x y z)) (if x y z)))

(defthm prv2-then-pass-shift
  (equal (prv2 (if x t (hide z))) (if x t (prv z)))

(defthm prv2-if-gen-prv
  (equal (prv2 (if x (prv3 y) z)) (prv2 (if x (prv y) z))))

(defthm prv-evaporates (equal (prv x) x))

(defthm prv-case-split
  (implies (sieve (case-split C))
           (equal (prv x) (prv2 (if C (prv3 x) (hide x))))))

(in-theory (disable prv prv2 prv3))
```

**Figure 5: Case Splitter Rewrite Rules**

It remains for us to describe the `case-split` filter function. We first note that the user may modify this (or any) filter function without affecting the soundness of the subsequent results from KAS. In the current implementation, the `case-split` filter function attempts to select the `if` test that will most quickly reduce the resulting terms. The `case-split` filter traverses the node `(prv` $\alpha$`)` and selects the `if` test `(equal` *lhs rhs*`)` such that *lhs* has the greatest number of weighted occurrences in `(prv` $\alpha$`)` in comparison to the *lhs* of the other `if` tests – weighted by proximity to the root node `(prv` $\alpha$`)`. The code defining this search uses the user stobj provided to all filter functions to store a hash-table used to record and lookup weighted counts from the traversal of the `(prv` $\alpha$`)` node. This heuristic is a first cut and subsequent efforts will certainly lead to more efficient and robust heuristics, but this simple heuristic has proven sufficient for the current applications of KAS, and specifically the pipeline application in Section 4.1.

The case-splitting rules work in tandem with a separate set of rules that extract the failing case of a failed proof attempt. This failing case – the conjunction of the tests encountered along a path to an irreducible non-`T` node – is reported to the user. These rules are triggered by the `gfl` function, which tags the node as a failure for KAS (using the `fail` operator) and then proceeds to dive through the `if` term resulting from `prv` and prints the if-tests and leaf node along the first failing path that it finds using the filter function `report-to-cw`, which simply prints a term to the comment-window and returns `T`. These rules are provided in Figure 6. The `defthmk` macro will take a term $\alpha$ to be proven and will call the KAS rewriter on `(gfl (prv` $\alpha$`))`.

## 4.1 Case Splitting Application: Pipeline

We use a simple pipeline example derived and modified from the pipeline in [4]. The goal of this example is to demonstrate the capacity and limitations of KAS rewriting. The

```
(defthm rfl-leaf-case
  (implies (sieve (report-to-cw leaf))
           (equal (rfl leaf x) x)))

(defthm rfl-if-tbr-case
  (implies (sieve (report-to-cw tst))
           (equal (rfl (if tst tbr fbr) x)
                  (rfl tbr x))))

(defthm rfl-if-fbr-case
  (implies (and (sieve (non-nilp tbr))
                (sieve (report-to-cw (not tst))))
           (equal (rfl (if tst tbr fbr) x)
                  (rfl fbr x))))

(defthm gfl-creates-rfl (equal (gfl x) (fail (rfl x x))))
(defthm gfl-reduce-t    (equal (gfl t) t))

(in-theory (disable gfl rfl))
```

**Figure 6: Failure Reporting Rewrite Rules**

```
(defun ma-matches-isa (x)
  (if (commit x)
      (equal (rep (ma x)) (isa (rep x)))
    (and (equal (rep (ma x)) (rep x))
         (< (rank (ma x)) (rank x)))))

(defun maX4 (m) (ma (ma (ma (ma (ma (flush m)))))))
(defun maX5 (m) (ma (maX4 m)))
(defun maX6 (m) (ma (maX5 m)))
(defun maX7 (m) (ma (maX6 m)))
(defun maX8 (m) (ma (maX7 m)))

(defthmk maX4-proof (ma-matches-isa (maX4 m)))
(defthmk maX5-proof (ma-matches-isa (maX5 m)))
(defthmk maX6-proof (ma-matches-isa (maX6 m)))
(defthmk maX7-proof (ma-matches-isa (maX7 m)))
(defthmk maX8-proof (ma-matches-isa (maX8 m)))

(defthmk ma-proof (ma-matches-isa (ma (ma (ma (ma x))))))
```

**Figure 7: Stuttering Refinement**

pipeline example is simple enough to comprehend quickly while still being representative of the case analysis often required in analyzing real systems. Additionally, the case analysis for the pipeline example explodes quickly as you add more steps of the system, which provides a good relative comparison between ACL2 simplification and KAS simplification. The pipeline example is included in the supporting materials.

The simple pipeline consists of the five stages: instruction fetch, decode, execute, write to memory, and write to register file. The goal is to demonstrate a stuttering refinement[13] between the pipeline and a single-step instruction set architecture or `isa`. The state of the `isa` is composed of an instruction memory, program counter, register file, and data memory. The pipeline updates the program counter in the fetch phase, but updates the data memory and register file in later stages. Further, the pipeline microarchitecture `ma` may stall an instruction if the register destination for a preceeding instruction matches one of the register sources for a succeeding instruction, which can lead to a bubble in the pipeline. In principle, we would like to verify that a step of `ma` matches a step in `isa` but, because of these bubbles, we have to allow for stuttering. Thus, the goal becomes to prove the function `ma-matches-isa` in Figure 7 where, for committed states, the `ma` step will match the `isa` step, and otherwise will stutter with a strictly decreasing natural-valued `rank` function.

The `rep` function maps the `ma` state to a corresponding `isa` state and is defined essentially as producing an `isa` state with program counter, data memory, and register file captures just before they are updated in the `ma` state. Thus, the program counter is stored for 4 `ma` steps and the data memory is stored for 1 `ma` step before being mapped to an `isa` state through `rep`.

The standard stuttering refinement proof would normally involve the definition, assumption, and proof of an invariant on the `ma` states. Following an approach similar to [4], we will instead prove that the refinement holds for a certain number of steps from a "flushed" state – the `flush` function merely clears out the pipe stage valid bits in a given `ma` state. The theorem `ma-proof` is a proof of the refinement in four `ma` steps from an arbitrary state with no need for an invariant. Still, our goal in this example is to keep the definition of what we are proving simple and also provide some metric for capacity of KAS rewriting in comparison to ACL2 and other tools that could be applied to this type of problem.

For ACL2 simplification, the `maX4-proof` is dispatched immediately, the `maX5-proof` takes several minutes, but `maX6-proof` blows up rapidly and can take several hours to prove (the author did not let the proof finish). For KAS, each of the proofs for `maX4-proof` through `maX7-proof` are take no more thant a few seconds to prove. The proof for `maX8` jumps up to 2-3 minutes and `maX9` takes hours. Further, the statistics provided by KAS demonstrate how rapidly the cases explode with each successive `ma` step. The `maX7` proof requires about 15 thousand promoted nodes and a few million transient nodes, while the `maX8` proof requires about 700 thousand promoted nodes and more than a hundred million transient nodes – please note that this is the number of transient nodes created throughout the entire proof; the total number of transient nodes created at any given time during the proof is only in the tens of thousands.

Thus, for problems that require significant case analysis, KAS can be far more efficient than ACL2 simplification. Additionally, for problems that can be encoded in decidable theories, KAS simplification can take more time than efficiently implemented decision procedures. It is worth mentioning that we purposely made no attempt in this example to try to control the terms that were being expanded in successive `ma` steps. Indeed, with some additional theorems, you could control how `ma` steps were expanded and reduce the case explosion considerably, but this would not be as effective in demonstrating KAS operation in comparison to ACL2 simplification and efficient decision procedures. The interested reader is encouraged to experiment with this simple pipeline example to better understand how KAS operates – it is particularly illustrative to change the definition of `ma` to create a failed proof and observe the resulting output from KAS.

## 5. CONCLUSIONS AND CURRENT WORK

We have presented the architecture and definition of a new simplifier based on optimized, ordered, inside-out, conditional rewriting with extensions for user-defined heuristics and decision procedures. This simplifier is termed KAS and its implementation affords efficient construction and manipulation of a large number of nodes and rewrite rule applications. KAS provides efficient memory management, unique construction of promoted nodes, and contextual memoization. The KAS simplifier is written in less than four thousand lines of ACL2 source code and this includes code for interfacing with ACL2 and various supporting definitions and macros. Further, the definition of KAS is structured to support a subsequent formal mechanical proof of its soundness in the ACL2 theorem prover.

We are currently attempting to integrate our work on automating invariant discovery[9] into KAS as a set of rewrite rules and filter functions. The straightforward integration of these efforts is complicated by the use of compiled functions in the invariant discovery process. In addition, the previous work used explicit state search; a more symbolic approach may be feasible using KAS. We are also working on a proof of soundness of KAS in ACL2. The basic outline of the soundness proof is to first prove a "model" of KAS is sound under the assumption that all rewrite rules currently in the world are valid, and then prove that KAS is equivalent to this "model". Defining and proving sufficient invariants of the "model" and KAS itself will be significant components in the work. In addition, we are always looking at ways to improve the efficiency of KAS operation and provide more examples of its application. Of course, any potential optimization or change in code must be weighed against the costs of additional complexity and potential effects on the predictability, soundness, or completeness of KAS operation. Throughout the development of KAS, many optimizations and features have been considered and even implemented, only to be removed after evaluation demonstrated that the benefits were not sufficient.

## 6. REFERENCES


[1] H. Barendregt, M. C. J. D. van Eekelen, J. R. W. Glauert, J. R. Kennaway, M. J. Plasmeijer, and M. R. Sleep. Term graph rewriting. In Proceedings of the European Workshop on Parallel Architectures and Languages, volume 259 of Lecture Notes in Computer Science, pages 141-158, Berlin, 1987. Springer-Verlag.

[2] R. Bryant. Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers, C-35, 8:677-691, 1986.

[3] R. E. Bryant and S. K. Lahiri and S. A. Seshia. Modeling and Verifying Systems using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions. In Proceedings of 14th International Conference on Computer-Aided Verification (CAV), LNCS 2404, 2002.

[4] P. Manolios and S. K. Srinivasan. Automatic Verification of Safety and Liveness for XScale-Like Processor Models Using WEB Refinements. DATE 2004, ACM-IEEE Design, Automation, and Test in Europe, 2004.

[5] P. Manolios and S. K. Srinivasan. A Parameterized Benchmark Suite of Hard Pipelined-Machine-Verification Problems. CHARME 2005, the 13th Advanced Research Working Conference on Correct Hardware Design and Verification Methods, 2005.

[6] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In Proceedings of the 38th Design Automation Conference (DAC'01), June 2001.

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

[8] N. Shankar and H. Rue. Combining Shostak theories. In Sophie Tison, editor, International Conference on Rewriting Techniques and Applications (RTA '02), Volume 2378 of Springer-Verlag Lecture Notes in Computer Science, pages 1–18, Copenhagen, Denmark, July 2002.

[9] S. Ray and R. Sumners. Combining Theorem Proving with Model Checking through Predicate Abstraction. IEEE Design & Test of Computers, volume 24(2), March-April 2007, pages 132-139. IEEE Computer Society.

[10] H. Rue and N. Shankar. Deconstructing Shostak. In 16th Annual IEEE Symposium on Logic in Computer Science, pages 19–28, IEEE Computer Society, Boston, MA, July 2001.

[11] R.E. Shostak. Deciding Combination of Theories. J. of the ACM, 31(1):1-12, 1984.

[12] J.P.M. Silva, K.A. Sakallah, Conflict analysis in search algorithms for propositional satisfiability. Technical Reports, Cadence European Laboraties, ALGOS, INESC, Lisboa, Portugal, May 1996.

[13] R. Sumners. An Incremental Stuttering Refinement Proof of a Concurrent Program in ACL2. In Kaufmann, M., Moore, J.S., eds.: Second International Workshop on ACL2 Theorem Prover and Its Applications, Austin, TX (2000)

[14] R. Sumners. Ph.D. Dissertion. Chapter 6. The University of Texas at Austin, 2005.

[15] H. Zhang. SATO: A decision procedure for propositional logic. Association for Automated Reasoning Newsletter, 22:1–3, March 1993. 13