Synthesis of Propositional Satisfiability Solvers

Douglas R. Smith
Stephen J. Westfold
Kestrel Institute
Palo Alto, CA, 94304 USA
{smith,westfold}@kestrel.edu
April 2008

1 Introduction

Development of system software by refinement has been extensively studied, and few larger scale case studies has been carried out. Most refinement approaches havse relied on manual construction of ad-hoc non-reusable refinements that are subjected to post-hoc verification. The problem with this approach is that the incremental cost of adapting to changing requirements is too high for this to be practical. Our belief is that refinement technology will only become feasible when most, if not all, refinements are generated correctly rather than constructed manually.

We recently carried out a case study of the automated synthesis of high-performance solvers for the propositional satisfiability problem (SAT). In the end we generated a small family of solvers. With the exception of the introduction of control heuristics, which is a manual step, all refinements were automatically generated using representations of design knowledge in a library. The library design knowledge had large overlaps with the knowledge used to generate high-performance transportation scheduling algorithms and others.

The fundamental result of this project is that we are able to recapitulate many of the key design features of a modern DPLL SAT solver using abstract and reusable design concepts: the Global Search and Constraint Propagation algorithm paradigms, expression simplification, finite differencing, and datatype refinement. The advantages of this approach include (1) the abstraction, formalization, and transfer of ideas from one problem domain to another, (2) the generation of both proof-of-correctness (certification evidence) and code in tandem, and (3) increased automation in the development process, leading to faster development and evolution of high-performance and high-assurance code. The insights gained from taking a more abstract and formal approach led to the discovery of the Sequent-based SAT algorithm.

During the project we demonstrated speedups of over 6 orders-of-magnitude improvement (≈1,300,000x speedup; see Figures 3 and 4). We discovered a new SAT algorithm that is not based on the DPLL foundation, but instead draws on our experience with high-performance scheduling algorithms [15, 16]. We subsequently found that Purdom [11] had published a very similar algorithm although it has not been developed with the latest SAT techniques.

2 SATware Derivations

The formal/mechanized derivation of SAT solvers is particularly challenging for the following reason. For many common problems, the problem constraints are given in the problem specification. Propositional Satisfiability, however, takes *constraints as input data* (i.e. the clauses) – its specification has a meta-constraint that the input clauses must all be satisfied. This level of indirection leads to some interesting challenges in the derivation process, which we have attempted to make as simple and transparent as possible.

2.1 Satisfiability Domain Theory

A specification for the propositional satisfiability domain builds up the vocabulary for specifying the SAT problem and for reasoning about it. In this section we give highlights of a SAT domain theory that supports DPLL-like algorithms. Only some of the key definitions and theorems will be presented here. The complete domain theory may be found in CNF.sw and related files.

The task of a SAT algorithm is essentially to analyze a given boolean function for its satisfiability; i.e. to decide the existence of inputs that give it the value `true`. Interestingly, SAT is rarely treated as a second-order function (taking a function as input). Instead it is treated as a kind of meta-function that operates on the *representation* of a function. The most common representation is as a formalization of conjunctive normal form (CNF), although many others are possible.

```
type Variable
type Literal   = | Pos Variable | Neg Variable
type Clause    = Set Literal
type CNF       = Map(Nat,Clause)
type Valuation = Map(Variable, Boolean)
```

We give semantics to the CNF representation by means of an interpreter, `evalCNF`, that evaluates the representation for a given valuation of the variables.

```
op evalLiteral (lit:Literal, vm:Valuation):Boolean =
   case lit of
     | Pos v -> TMApply(vm,v)
     | Neg v -> ~(TMApply(vm,v))

op evalClause (c:Clause, vm:Valuation):Boolean =
    set_fold (false)
             (fn(lit:Literal, b:Boolean)-> evalLiteral(lit,vm) || b)
             (c)
```

```
op evalCNF(p:CNF, vm:Valuation):Boolean =
   set_fold (true)
            (fn(clauseIndex:Nat, b:Boolean)->
             evalClause(TMApply(p,clauseIndex), vm) && b)
            (domain(p))

op satisfiable(p:CNF):Boolean =
    ex(vm:Valuation)(varsOf(p)=domain(vm) && evalCNF(p,vm)))
```

The SAT problem is expressed as a function (from CNF to an Valuation, if one exists). Rather than supply a definition, we specify SAT by means of an input/output predicate (output condition) that provides the minimal constraints on acceptable implementations.

```
Type Option a = | None | Some a

op SAT : CNF -> Option Valuation

axiom SAT_spec is
   fa(p:CNF)
      case SAT(p) of
        | Some v -> evalCNF(p,v)
        | None   -> ~satisfiable(p)
```

2.2 Algorithm Development: Global Search with Constraint Propagation

Global Search (GS) is an algorithm abstraction that generalizes binary search, backtracking, and branch-and-bound. Global Search (GS) with constraint propagation explains the algorithmic structure of virtually all complete SAT solvers. It is the method of choice for exactly solving most NP-hard problems. The basic idea of Global Search is to represent and manipulate sets of candidate solutions. The principal operations are to *extract* candidate solutions from a set and to *split* a set into subsets. Derived operations include various *filters* which are used to eliminate sets containing no feasible or optimal solutions. Global search algorithms work as follows: starting from an initial set that contains all solutions to the given problem instance, the algorithm repeatedly extracts solutions, splits sets, and eliminates sets via filters until no sets remain to be split. The process is often described as a tree (or DAG) search in which a node represents a set of candidates and an arc represents the split relationship between set and subset. The filters serve to prune off branches of the tree that cannot lead to solutions.

The sets of candidate solutions are often infinite and even when finite they are rarely represented extensionally. Thus global search algorithms are based on an abstract data type of intensional representations called *space descriptors*. In addition to the extraction and splitting operations mentioned above, the type also includes a predicate *satisfies* that determines when a candidate solution is in the set denoted by a descriptor.

Constraint Propagation(CP)is a technique commonly used with Global Search . Its effect is to iteratively tighten the representation of a subspace descriptor while preserving existence of a solution in general. The classic DPLL algorithm makes use of two propagation rules: the Unit Rule and the Pure Literal Rule [4, 3].

DPLL is based on representing sets of candidate solutions by partial valuations of variables, where a partial valuation denotes the set of all of its possible extensions. We formalize partial valuations as finite maps from Variables to 4-valued logic which forms a partial order according to

$$\begin{tikzcd}
& \texttt{Top} & \\
\texttt{False} \arrow[ur] & & \texttt{True} \arrow[ul] \\
& \texttt{Bot} \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

```
type Logic4 = | Bot | False | True | Top

def Logic4Le (p:Logic4, q:Logic4): Boolean =
  case p of
    | Bot -> true
    | False -> (case q of
                  | False -> true
                  | Top   -> true
                  | _     -> false)
    | True -> (case q of
                  | True -> true
                  | Top  -> true
                  | _    -> false)
    | Top  -> (case q of
                  | Top  -> true
                  | _    -> false)
```

Intuitively, `Bot` represents an undefined or unknown value, and `Top` represents an overdefined or inconsistent value. More specifically, Logic4 forms a Boolean Algebra with meet operator `Meet4`, join operator `Join4`. We interpret the operators of Propositional Logic in terms of `And4`, `Or4`, and `Not4` in Logic4. Interestingly, these are different from the lattice operations in Logic4. See Logic4.sw for more detail.

A partial valuation will be represented by a map from variables to Logic4 values. The use of the `Top` element models situations in which we infer inconsistent values for a variable (e.g. during

Boolean Constraint Propagation).

```
  type Valuation4 = Map(Variable, Logic4)
```

To simplify the presentation, let `m(a)` denote the application of a map `m` to a domain element `a`, where `m(a)` returns `Bot` if `a` is not in the domain of `m`, and the value associated to `a` otherwise.

With this richer notion of valuation, we give homomorphic analogues to `evalLiteral`, `evalClause`, `evalCNF`:

```
op eval4Literal (lit:Literal, vm:Valuation4):Logic4 =
  case lit of
    | Pos var -> vm(var)
    | Neg var -> Not4(vm(var))

op eval4Clause (c:Clause, vm:Valuation4):Logic4 =
   set_fold (False)
            (fn(lit:Literal, b4:Logic4)-> Or4(eval4Literal(lit,vm),b4))
            (c)

op eval4CNF (p:CNF, vm:Valuation4):Logic4 =
   set_fold (True)
            (fn(ci:Nat, b4:Logic4)-> And4(eval4Clause(p(ci),vm),b4))
            (domain(p))

theorem homomorphism_of_evalLiteral is
   fa (l:Literal, vm: Valuation)
   BooleantoLogic4(evalLiteral(l,vm)) = eval4Literal(l,ValtoVal4(vm))

theorem homomorphism_of_evalClause is
   fa (c:Clause, vm: Valuation)
   BooleantoLogic4(evalClause(c,vm)) = eval4Clause(c,ValtoVal4(vm))

theorem homomorphism_of_evalCNF is
   fa (p:CNF, vm: Valuation)
   BooleantoLogic4(evalCNF(p,vm)) = eval4CNF(p,ValtoVal4(vm))
```

It is useful to introduce a notion of relative satisfiability

```
op satisfiableRel(p:CNF,pm:Valuation4):Boolean =
   ex(vm:Valuation4)(Valuation4Le(pm,vm) && evalCNF(p,vm)))
```

Map theory provides a single-point update operator and we define a binary composition operator on Valuation4.

```
type Map a b = | empty_map | update (Map a b)*a*b

op compose(val1:Valuation4, val2:Valuation4): Valuation4 =
   set_fold (val1)
            (fn(v:Variable, newval:Valuation4) ->
               update(newval, v, Join4(val1(v),val2(v))))
            (domain(val2))
```

When the range type of a Map is a partial order, we can naturally define a partial order on the Map type:

```
op Valuation4Le (val1:Valuation4, val2:Valuation4): Boolean =
   set_fold (true)
            (fn (v:Variable, b:Boolean) ->
              Logic4Le(val1(v),val2(v)) && b))
            (domain(val1))
```

For purposes of calculation in this report, we use the more concise infix notation

$val1 \sqsubseteq val2$ instead of `Valuation4Le(val1,val2)`

and we use

$val1 \oplus val2$ instead of `compose(val1,val2)`.

Theorem: Monotonicity of Valuations with respect to composition:

$$m \sqsubseteq n \ \wedge \ p \sqsubseteq q \implies p \oplus m \sqsubseteq q \oplus n \tag{1}$$

Theorem: Preservation of upper bound under map composition

$$\bigwedge_i (m_i \sqsubseteq n) \ = \ (\bigoplus_i m_i) \sqsubseteq n \tag{2}$$

Moreover, we have the following theorems:

Theorem: `evalCNF` and `eval4CNF` are monotone in valuations

$$\forall (m, n : Valuation)\ (m \sqsubseteq n \implies (evalCNF(p, m) \implies evalCNF(p, n))) \tag{3}$$

$$\forall(m,n:Valuation4)\ (m \sqsubseteq n \implies (eval4CNF(p,m) \implies eval4CNF(p,n))) \tag{4}$$

Furthermore, in this project, we discovered a general set of lattice-based laws that are invaluable in deriving and reasoning about SAT algorithms (and other kinds of algorithms [16]). These laws are laid out in detail in Appendix A. By way of introduction we discuss two here.

If F is a monotone function from a preorder $\langle A, \preceq\rangle$ to a lattice $\langle Boolean, \wedge, \vee, \Rightarrow\rangle$ (i.e. F is a predicate), then we have

Monotone $F : \langle A, \preceq\rangle \to \langle Boolean, \wedge, \vee, \Rightarrow\rangle$			
2.1	$\bigvee_{a:A \mid a \preceq \hat{a}} F(a) \ = \ F(\hat{a})$	$\bigwedge_{a:A \mid \check{a} \preceq a} F(a) \ = \ F(\check{a})$	2.2

As a special case if F is a proposition then we have

Monotone $F : \langle Boolean, \Rightarrow\rangle \to \langle Boolean, \wedge, \vee, \Rightarrow\rangle$			
3.1	$\exists(a : Boolean)\ F(a) \ = \ F(true)$	$\forall(a : Boolean)\ F(a) \ = \ F(false)$	3.2

There are dual laws for when F is antimonotone. Note that 3.1 is the basis for the Pure Literal Rule, and 3.2 is the basis for the analog to PLR in a QBF setting where a variable is universally quantified [17]. This insight is elaborated in Appendix B.

2.3 Global Search

We outline the pertinent aspects of Global Search theory. More detail may be found in [13, 16] or a book that Smith is writing. Global Search theory has two parts: a program scheme and a specification of the free operators of the scheme. The axioms are sufficient to prove the correctness of the scheme. If we can construct an interpretation of the free operators that satisfy all the axioms, then we can instantiate the scheme with confidence that a correct algorithm results.

The purpose of Global Search theory is to guide the instantiation of the scheme, according to the axioms, so that a correct-by-construction algorithm is obtained.

A Global Search scheme that uses chronological backtracking (versus conflict-directed backtracking) together with pruning and propagation is:

```
GlobalSearchPropagation = spec
  import GlobalSearchPropagationTheory

  (* f is the top level of the Global Search algorithm to find one
  feasible solution. *)
  def f(x:D): Option R =
    let r0 : Option State = Propagate(InitialState(x)) in
    case r0 of
      | None -> None
      | Some r -> GS(x,r)

  (* GS explores the search tree using a depth-first strategy.  *)
  def GS (x:D, r:State | Phi(r) && Psi(r)=r && Xi(r)=r) : Option R =
    let sol: Option R = Extract(r) in
    case sol of
      | Some z | O(x,z) -> Some z
      | None   -> GSAux(x,r,Subspaces(r))

  (* GSAux explores the children of a space.  *)
  def GSAux (x:D, r:State, rs:List SplitInfo
            | Phi(r) && Psi(r)=r && Xi(r)=r) : Option R =
    case rs of
      | []   -> None
      | hd::tl -> case Propagate(Split(r,hd)) of
                    | None -> GSAux(x,r,tl)
                    | Some newr -> (case GS(x,newr) of
                                      | None -> GSAux(x,r,tl)
                                      | Some r -> Some r)

  op Propagate : State -> Option State
  axiom Propagate_computes_fixpoint is
    fa(s0:State,s1:State)
      (case Propagate(s0) of
         | Some s1 -> RefinesTo(s0,s1)
                      && Psi(s1)=s1 &&  Xi(s1)=s1 && Phi(s1)
         | None -> ~(ex(z:R)(Satisfies(z,s0))))

  theorem correctness_of_GS is
   fa(x:D) (case f(x) of
              | Some z -> O(x,z)
              | None -> ~(ex(z)O(x,z)))
end-spec
```

```
DROfPartial = spec
 type D                          % input type
 type R                          % output type
 op O : D * R -> Boolean         % output condition
 op f : D -> Option R            % problem solver
 axiom correctness_of_f is
   fa(x:D) case f(x) of
            | Some z -> O(x, z)
            | None   -> fa(y:R) ~(O(x,y))
 end-spec

GlobalSearchTheory = spec
  import DROfPartial
  type State
  op InitialState : D -> State

  (*  Satisfies(z,s) means that z is "in" the space denoted by s *)
  op Satisfies : R * State -> Boolean

  (*  All feasible solutions are "in" the initial space ... *)
  axiom initial_space_contains_all_solutions is
     fa(x:D,z:R)(O(x,z) => Satisfies(z,InitialState(x)))

   (* ... and the elements in a space are either directly extractable
   or can be extracted after splitting into subspaces.*)
  op Extract : State -> Option R
  type SplitInfo
  op Subspaces : State -> List SplitInfo
  op Split : State * SplitInfo -> State

   axiom definition_of_satisfies is
     fa(x:D, s:State, z:R)
      Satisfies(z, s) = ((Extract(s) = Some z)
                        or (ex (si:SplitInfo)
                            (member(si, Subspaces(s))
                             && Satisfies(z, Split(s,si)))))

   op Phi : State -> Boolean
   axiom necessary_filter_on_feasible_subspaces is
     fa(x:D,z:R,s:State)(Satisfies(z,s) & O(x,z) => Phi(s))

 end-spec
```

which is extended with pruning and propagation operators:

```
GlobalSearchPropagationTheory = spec
  import GlobalSearch#GlobalSearchTheory
  import translate ../Math/PartialOrder by
      {A +-> State, POle   +-> RefinesTo}

  axiom InitialState_is_bottom is
    fa(x:D, s:State) RefinesTo(InitialState(x), s)

  axiom Satisfies_implies_RefinesTo is
     fa(x:D,s:State,u:State,z:R)
       (Extract(u) = Some z
        && Satisfies(z, s)
        => RefinesTo(s, u))

  axiom Satisfies_is_antimonotone is
      fa(x:D, s:State, s1:State, z:R)
      RefinesTo(s, s1) => (Satisfies(z,s1) => Satisfies(z,s))

  def [a] monotone? (f:a->a, le:a*a->Boolean):Boolean =
      fa(x:a,y:a)( le(x,y) => le(f(x),f(y)) )

  (* Psi is a monotone function on spaces that adds necessary
  constraints to the current space.  By construction it preserves all
  feasible solutions. *)
  op Psi : State -> State
  axiom characterization_of_necessary_propagator_Psi is
    monotone?(Psi, RefinesTo) &&
    (fa(x:D,r:State,z:R)(Satisfies(z,r) & O(x,z) => Satisfies(z,Psi(r))))

  (* Xi is a monotone function that generates a consistent refinement
  of a space. By construction it preserves the existence of feasible
  solutions. *)
  op Xi : State -> State
  axiom characterization_of_consistent_refinement_Xi is
    monotone?(Xi, RefinesTo) &&
    (fa(x:D,r:State) (ex(z:R)(Satisfies(z,r)     && O(x,z))
                  = (ex(z:R)(Satisfies(z,Xi(r)) && O(x,z)))))

 end-spec
```

2.3.1 Constructing a Global Search Theory Interpretation

To instantiate the Global Search scheme, we need to first develop an interpretation for each of the scheme operators. If we can interpret the operators such that the axioms of Global Search theory hold, then the scheme correctness theorem applies to the instantiated scheme.

The specification of SAT gives us the interpretation of D (input type), R (output type), and O (output condition). As noted earlier, DPLL is based on representing sets of candidate solutions by a map from variables to 4-logic. Each such map denotes the set of all of its possible extensions. The partial order on 4-logic naturally lifts to a partial order on valuations. This partial order on valuations is the refinement order in Global Search . We define `State` as a tuple

```
type State = {prop:CNF,varVal:Valuation4 | domain(varVal)=varsOf(prop)}
```

which packages the current Valuation together with the input proposition.

The `mkInitial`function is defined by the bottom element of `Valuation4`.

```
def mkInitialState(p:CNF):State =
 {prop   = p,
  varVal = mapFrom(varsOf(p), fn(var:Variable)->Bot)}
```

`Subspaces` is defined for free by the single constructor of Maps: we choose a variable (according to a heuristic) and then consider/generate all possible updates of the current `Valuation4`.

In summary, we embed the output type of the problem homomorphically into a finite lattice of Maps, and straightforwardly fill out the basic parts of an interpretation from Global Search theory into SAT. Next, we calculate further parts of the interpretation.

2.3.2 Calculating the Pruning Test: Phi

In Global Search theory, the pruning test is characterized by

```
 axiom characterization_of_necessary_pruning_test_Phi is
   fa(x:D,s:State)( ex(z:R)(Satisfies(z,s) & O(x,z)) => Phi(s))
```

Note that this axiom doesn't provide a definition. Instead it characterizes pruning tests as a necessary condition that subspace `s` contains a feasible solution. `Phi` is tested on each subspace (node of the backtrack tree). If it evaluates to false, then by the contrapositive of the axiom, there does not exist a feasible refinement of `s`, so `s` can be eliminated from further exploration.

Not only does the axiom provide a general characterization of pruning mechanisms in general, but it also suggests how to derive them. We start by instantiating the antecedent of the axiom using the interpretation built so far. Then we reason forwards, weakening until we derive an expression in terms just of `s`.

For SAT, we calculate as follows:

assume: $p : CNF, st : State$
$\quad\quad \wedge\, pm = st.varVal$
source: $\exists(vm : Valuation)(pm \sqsubseteq ValtoVal4(vm) \;\wedge\; evalCNF(p, vm))$

$\equiv$ { apply homomorphism-of-evalCNF theorem }

$$\exists(vm : Valuation)(pm \sqsubseteq ValtoVal4(vm) \;\wedge\; eval4CNF(p, ValtoVal4(vm)) = True)$$

$\Longrightarrow$ { weakening $=$ to $\sqsubseteq$ in 4-Logic }

$$\exists(vm : Valuation)(pm \sqsubseteq ValtoVal4(vm) \;\wedge\; eval4CNF(p, ValtoVal4(vm)) \sqsubseteq True)$$

$\equiv$ { Quantifier Elimination Law -2.1 }

$$eval4CNF(p, pm) \sqsubseteq True.$$

That is, the current value of the proposition with respect to the partial valuation `s.varVal` must be `Bot` or `True`. Two comments are in order. First, this is the obvious pruning test that any good programmer would come up with, but the point is that we can calculate it formally. We obtain a proof of its correctness as a by-product of its derivation. Second, as stated, the pruning test is rather inefficient, since the straightforward implementation would repeatedly evaluate the proposition under small changes to the partial valuation. An efficient implementation will be derived later using a technique called Finite Differencing (see Section 2.4).

2.3.3 Calculating the Necessary Propagator: Psi

Constraint Propagation is based on a propagation operator (propagator) that tightens the representation of a subspace. By iterating a propagator to a fixpoint, we can often dramatically reduce the search space. In the general case, propagators preserve existence of feasible solutions (detailed in the next subsection). However, almost all propagators in the literature have a stronger property: they eliminate some infeasible solutions while preserving *all* feasible solutions. We call these *necessary propagators* and the Unit Rule is an example.

In Global Search theory, a necessary propagator is characterized by

```
axiom characterization_of_necessary_propagator_Psi is
  monotone?(Psi, RefinesTo) &&
  fa(x:D,r:State,z:R)(Satisfies(z,r) & O(x,z) => Satisfies(z,Psi(r)))
```

Again, note that this axiom doesn't provide a definition. Instead it characterizes a necessary propagator as a monotone function that refines `r` and preserves all feasible solutions. As with pruning tests, the form of the axiom suggests how to derive necessary propagators. We start by instantiating the antecedent of the second conjunct (using the interpretation built so far), then we reason forwards towards an expression of the form in the consequent. If we can derive a definition for `Psi` in this way, then we can check that the first conjunct in the axiom holds; i.e. that `Psi` is a monotone function.

For SAT, we calculate as follows:

assume: $p : CNF, vm : Valuation, st : State$
$\wedge\, pm = st.varVal$
source: $pm \sqsubseteq ValtoVal4(vm) \;\wedge\; evalCNF(p, vm)$

$\equiv$ { unfold *evalCNF* (we leave $pm \sqsubseteq ValtoVal4(vm)$ as a tacit assumption) }

$\bigwedge_{c \in p} evalClause(c, vm)$

$\equiv$ { unfold *evalClause* }

$\bigwedge_{c \in p} \bigvee_{lit \in c}\; evalLiteral(lit, vm)$

$\Longrightarrow$ { in a disjunction, if all but one of the disjuncts are false, then the remaining one must be true }

$$\bigwedge_{c \in p} \big(\bigvee_{lit \in c} \bigwedge_{lit' \in c} (lit' \neq lit \implies \neg evalLiteral(lit', vm)) \implies evalLiteral(unit(c, vm), vm)\big)$$

where

$$unit(c, vm) = the(lit)(lit \in c \ \wedge \ evalLiteral(lit, vm))$$

$\equiv$ { embedding into Logic4 using homomorphism-of-evalLiteral }

$$\textstyle\bigwedge_{c \in p}(\bigvee_{lit \in c} \ \bigwedge_{lit' \in c} (lit' \neq lit \implies eval4Literal(lit', ValtoVal4(vm)) = \texttt{False}) \\ \implies eval4Literal(unit4(c, ValtoVal4(vm)), ValtoVal4(vm)) = \texttt{True}\big)$$

where

$$unit4(c, vm4) = the(lit)(lit \in c \ \wedge \ eval4Literal(lit, vm4) = True)$$

$\equiv$ { use Lemma $(eval4Literal(l, m) = b) \ = \ varmap(l, b) \sqsubseteq m)$ }

$$\textstyle\bigwedge_{c \in p} \big(\bigvee_{lit \in c} \ \bigwedge_{lit' \in c} (lit' \neq lit \implies varmap(lit', \texttt{False}) \sqsubseteq ValtoVal4(vm)) \\ \implies varmap(unit4(c, ValtoVal4(vm)), \texttt{True}) \sqsubseteq ValtoVal4(vm)\big)$$

where

$$\begin{aligned} varmap(l, b) \ = \ & case\ l\ of \\ & \quad \mid pos\ v \ \rightarrow \ \{v \mapsto b\} \\ & \quad \mid neg\ v \ \rightarrow \ \{v \mapsto \neg b\} \end{aligned}$$

$\implies$ { replace $ValtoVal4(vm)$ by pm twice, by polarity }

$$\textstyle\bigwedge_{c \in p} \big(\bigvee_{lit \in c} \ \bigwedge_{lit' \in c} (lit' \neq lit \implies varmap(lit', \texttt{False}) \sqsubseteq pm) \\ \implies varmap(unit4(c, pm), \texttt{True}) \sqsubseteq ValtoVal4(vm)\big)$$

$\implies$ { abstracting }

$$\textstyle\bigwedge_{c \in p} unit?(c, pm) \implies varmap(unit4(c, pm), \texttt{True}) \sqsubseteq ValtoVal4(vm)$$

where

$$unit?(c, pm) = \textstyle\bigvee_{lit \in c} \ \bigwedge_{lit' \in c} (lit' \neq lit \implies varmap(lit', \texttt{False}) \sqsubseteq pm)$$

$\equiv$ { applying Theorem (2) (Preservation of bound under map composition) }

$$\big(\textstyle\bigoplus_{c \in p, unit?(c, pm)} varmap(unit4(c, pm), \texttt{True})\big) \sqsubseteq ValtoVal4(vm).$$

which has the desired form, letting

$$Psi(p, pm) = pm \oplus \bigoplus_{\substack{c \in p, \\ unit?(c, pm)}} varmap(unit4(c, pm), \texttt{True})$$

or, in terms of `State`

$$Psi(st) = st.varVal \oplus \bigoplus_{\substack{c \in st.prop, \\ unit?(c, st.varVal)}} varmap(unit4(c, st.varVal), \texttt{True})$$

It is easy to check that Psi is monotone – see Appendix A.

2.3.4 Calculating a Consistent Refinement Propagator: Pure Literal Rule

As mentioned above, the most general class of propagators preserve the existence of feasible solutions. They are called Consistent Refinements since they refine the current space and preserve consistency (existence of solutions/models). The Pure Literal Rule in SAT is an instance.[1]

In Global Search theory, a Consistent Refinement propagator is characterized by

```
axiom characterization_of_consistent_refinement_Xi is
  monotone?(Xi, RefinesTo) &&
  (fa(x:D,s:State) (ex(z:R)(Satisfies(z,s)     && O(x,z))
                 = (ex(z:R)(Satisfies(z,Xi(s)) && O(x,z)))))
```

Note that, again, this axiom doesn't provide a definition. Instead it characterizes a Consistent Refinement `Xi` as a monotone function that refines `s` and preserves existence of feasible solutions. As with the pruning and necessary propagation tests, the form of the axiom suggests how to derive `Xi`. We start by instantiating the LHS of the second conjunct (using the interpretation built so far), and then perform equational reasoning towards an expression of the form in the RHS. If we can derive a definition for `Xi` in this way, then we can check that `Xi` is a monotone function.

For SAT, we calculate as follows:

[1] As an aside, refinement of formal specifications (the medium of our approach) is based on consistent refinement – at each stage of development we wish to construct a refinement of our current specification that preserves its consistency. Ultimately we wish to extract one of those models and express it in a programming language. So the Specware problem of refining specifications to code is also an instance of Global Search with consistent refinements.

assume: $p : CNF, pm : Valuation4$
source: $\exists(vm)(pm \sqsubseteq ValtoVal4(vm) \ \wedge \ evalCNF(p, vm))$

$=$ { by definition of *satisfiableRel* }

$$\mathit{satisfiableRel}(p, pm)$$

$=$ { CNF version of Quantifier Elimination Law (see notes below) }

$$\mathit{satisfiableRel}(p, pm \oplus \bigoplus_{monotone(p,v)} \{v \mapsto True\} \oplus \bigoplus_{antimonotone(p,v)} \{v \mapsto False\})$$

$=$ { unfolding def of *satisfiableRel* }

$$\exists(vm)(Xi(p, pm) \sqsubseteq ValtoVal4(vm) \ \wedge \ evalCNF(p, vm) = True)$$

where

$$Xi(p, pm) = pm \oplus \bigoplus_{monotone(p,v)} \{v \mapsto True\} \oplus \bigoplus_{antimonotone(p,v)} \{v \mapsto False\}).$$

Note: Recall the Quantifier Elimination law that was discussed earlier (and listed in the Appendix):

$$(\exists(a : Boolean)\, F(a)) \ = \ F(\mathit{true}) \qquad \text{if monotone(F,a)}$$

and its dual

$$(\exists(a : Boolean)\, F(a)) \ = \ F(\mathit{false}) \qquad \text{if antimonotone(F,a).}$$

This is a theorem about functions, but we want to apply to a CNF representation of a function. The result is

$$\forall(v)(monotone(p, v) \implies (\mathit{satisfiable}(p) = \mathit{satisfiableRel}(p, \{v \mapsto True\}))$$

and

$$\forall(v)(antimonotone(p, v) \implies (\mathit{satisfiable}(p) = \mathit{satisfiableRel}(p, \{v \mapsto False\}))$$

or, more generally,

$$\mathit{satisfiableRel}(p, pm) = \mathit{satisfiableRel}(p, pm \oplus \bigoplus_{monotone(p,v)} \{v \mapsto True\} \oplus \bigoplus_{antimonotone(p,v)} \{v \mapsto False\}).$$

It is easy to check that Xi is monotone – see Appendix A.

2.3.5 Conflict Analysis

In the CDB derivations, we implemented a version of the *decision strategy* for conflict analysis [18]. That is, the conflict reason is taken to be the subset of decision variables that lead to the conflict in the implication graph. We then backjump to the deepest decision level, above the current level, whose decision variable is in the conflict clause, and we can infer the negated form of the variable from the current decision level via the unit rule.

As a principle though, it is better to use as close a semantic approximation to the ideal reason as possible. In this situation, we want the weakest conjunction of literals that implies the contradiction (failed clause). Then, when we negate the conjunction, we obtain the strongest possible conflict clause, which can be used for backjumping and learning.

For this reason it seems best to use the *1UIP strategy* introduced in GRASP [8] and refined in zCHAFF [18]. The latter paper gave empirical evidence of the efficacy of 1UIP, but our more semantic justification provides, we think, a more reasoned rationale for its use. Current efforts seek to generalize conflict analysis and the 1UIP strategy to the Global Search theory, so that it can be applied mechanically to a variety of other problems.

2.3.6 Heuristics

Global Search defines the branching structure of a search, but does not prescribe the order in which to search it. The choice of which variable and which polarity to assign next is called the *heuristic* in SAT algorithms. Some easy heuristics are (1) to consider the variables in some predetermined order, and (2) random choice. There has been relatively little work on the nature of heuristics in general.

Here are some preliminary ideas about how one might go about deriving heuristics in a systematic way, much like we have derived the pruning and propagation operators. Heuristics are usually intended to improve the performance of the algorithm; a better heuristic leads to faster convergence on a solution. If we had a precise specification of the cost of a search process, we could begin to systematically transform it via *approximation rules*, supplementing the usual equational and inequational rules of the domain. Smith gave a sketchy example of this in his tutorial at HCSS.

The examples that we derived make use of two popular heuristics from the 1990's: MOMs (Maximum Occurences in Minimal size) and DLIS (Dynamic Largest Individual Sum). MOMs evaluates each open variable, counting the number of occurrences that it has in clauses of minimal size. It chooses to branch on that variable with the largest count. The rationale is based on the observation that DPLL-based algorithm spend most of their time in unit rule propagation. By creating as many unit clauses as quickly as possible, the intention is to speed up finding a contradiction or a complete solution. The DLIS heuristic aims to speed up search by choosing a variable that will have maximal impact on the overall proposition. For each open variable, it counts the number of positive and negative occurences in open clauses. In one variant, the variable with the largest sum is choosen, thereby touching a maximal number of clauses. In the positive branch the clauses with positive occurences are satisfied, and the clauses with negative occurences have one fewer literal.

We have structured the derivations so that it is easy to modify the heuristic or replace it with another. For example, in CDBmom.sw and CDBdlis.sw we have essentially the same derivation structure except that in CDBmom.sw the second step (`sat1`) introduces the MOMs heuristic versus DLIS. In our derivations, with all other design decisions being equal, the MOMs heuristic generally outperforms DLIS.

One of the most successful heuristics in state-of-the-art solvers is the VSIDS heuristic [18]. We did not implement this since its effectiveness comes when the SAT solver learns new clauses, which we did not get to in the current project.

2.4 Finite Differencing

The algorithm design calculations in the previous section have produced a correct, but not necessarily efficient design – there are usually many opportunities for performing optimizing transformations. This is a feature, not a flaw, in the design methodology. The derivational approach has a natural separation of concerns. Typically, it is the job of algorithm design to get a correct, high-level design in place. Then, we apply a sequence of optimizing transformations and refinements that realize the intrinsic capability of the algorithm. The two figures in Section 3 show the performance improvements that result from these optimizations.

Probably the single most important optimization/refinement technique is known as *context-sensitive simplification* (CSS) in which we simplify a program expression taking contextual properties into account. CSS is used as a primitive in many other transformations.

It often happens in algorithm design that an expensive expression needs to be repeatedly computed. The Finite Differencing transformation [10] is used to incrementally compute the value of the expression. In effect it trades some space (a new data structure) for time (expensive repeated computation is replaced by incremental updates of the data structure).

The Specware realization of FD is based on Coglio's Type Isomorphism Transformation [1]. The Isomorphism transformation is based on an isomorphism between two types, say D and D', given by isomorphisms $iso : D \to D'$ and $osi : D' \to D$. The transforms rewrite the whole specification, redefining all types and operators in terms of D'. For example, for each type such as $T = D * E$, a new type $T' = D' * E$ is introduced into the spec along with appropriate isomorphisms. For each operator $f : D \to D$, a new variant is introduced into the spec $f' : D' \to D'$ defined by

$$f'(d' : D) = iso(f(osi(d'))).$$

Various (context-sensitive) simplifications are applied to transform away references to the old types, operators, and isomorphisms, leaving the spec reformulated in terms of the new types and operators.

```
...
    $f(V)$
    ...

function $f(x : D)$
where $I(x)$
=   ...
    $E(x)$
    ...
    $f(U(x))$
    ...
```

$$\longrightarrow$$

```
...
    $f(V, E(V))$
    ...

function $f(x : D, c : C)$
where $I(x) \wedge c = E(x)$
=   ...
    $E(x)$
    ...
    f$(U(x), E(U(x)))$
    ...
```

Figure 1: Abstraction Operation underlying Finite Differencing

Finite Differencing is accomplished in the following way. In terms of Figure 1, we first introduce a new datatype $(D * C|fn(d,c) \rightarrow c = E(d))$ that is isomorphic to D. The Isomorphism transformation then transforms the whole specification, redefining all types and operators in terms of the new type. Then CSS is applied to simplify away references to the old types and operators, leaving the spec reformulated in terms of the new type and operators.

Performing Finite Differencing Our DPLL and CDB derivations create nine Finite Difference variables. We illustrate the transformation by focusing on one: the maintenance of the set of indices of unsatisfied clauses:

$$\{ci \mid ci \in domain(p) \ \wedge \ EvalClause(p(ci), pm) \sqsubseteq False\ \} \tag{5}$$

We will use Finite Differencing to create a new variable called `unsatCs:Set Nat` (short for unsatisfiedClauses) whose value is maintained equal to (5).

In the CDB derivation script, `unsatCs` is introduced in `sat5` via the new type `State''` which is isomorphic to the previously introduced type `State'` by `iso1` and `osi1`.

Lower down in `sat5` you will find the definition of unsatClauses into a function `mkUnsatClauses` that encapsulates (5):

```
op mkUnsatClauses(p:CNF, vm:Valuation4):Set Nat =
   set_fold (empty_set)
            (fn (ci:Nat, ucs:Set Nat) ->
                if unsatisfiedClause?(TMApply(p,ci),vm)
                  then set_insert(ci,ucs)
```

```
                 else ucs)
             (domain(p))

 op unsatisfiedClause?(c:Clause,vm:Valuation4):Boolean =
    Refines(EvalClause(c, pm), False)
```

This allows us to specify the semantics of `unsatCs` in `State'` by giving its invariant as a subtype property

$$unsatCs = mkUnsatClauses(st.prop, st.varVal) \tag{6}$$

in any state *st*. Near the definition for `mkUnsatClauses` in `sat5` you will find further support for the incremental computation of `unsatCs`. In the CDB algorithm there are only two operations that change the current valuation: *mkInitialState* and *updateState*. The essence of Finite Differencing is knowing how to maintain the invariant (6) under those two operations. We show the calculations next.

Calculating an initial value of `unsatCs`

```
assume: st.prop:CNF,
        st.varVal = mapFrom(varsOf(p), fn(v:Variable)->Bot)
source: unsatCs = mkUnsatClauses(st.prop,st.varVal)

 =   {substituting st.varVal}

    unsatCs = mkUnsatClauses(st.prop,mapFrom(varsOf(p), fn(v:Variable)->Bot))

 =    { unfolding mkUnsatClauses }

{ ci | ci in domain(p)
       && Valuation4Le(EvalClause(p(ci),
                      mapFrom(varsOf(p), fn(var:Variable)->Bot)), False) }

assume: st.prop:CNF,
        st.varVal = mapFrom(varsOf(p), fn (var:Variable) -> Bot)
source: unsatCs = mkUnsatClauses(st.prop,st.varVal)

=    {substituting st.varVal}

   unsatCs = mkUnsatClauses(st.prop,mapFrom(varsOf(p),
                                    fn (var:Variable) -> Bot))
=    {unfolding mkUnsatClauses}
```

```
   { ci | ci in? domain(p) &&
          Refines(Eval4Clause(p(ci),mapFrom(varsOf(p),fn (var:Variable) -> Bot)),
                  False)}

=    {focusing on Eval4Clause and unfolding it}

     ... set_fold (False)
           (fn(lit:Literal, b4:Logic4)->
                    Or4(eval4Literal(lit, mapFrom(varsOf(p),fn (var:Variable) -> Bot)),
                        b4))
                   (p(ci)) ...

=    {focusing on eval4Literal and unfolding it}

     ... case lit of
          | Pos var -> TMApply(mapFrom(varsOf(p),fn (var:Variable) -> Bot),var)
          | Neg var -> Not4(TMApply(mapFrom(varsOf(p),fn (var:Variable) -> Bot),var)) ...

=    {simplifying}

     ... Bot ...

=    {focusing back on Eval4Clause}

     ... set_fold (False)
                   (fn(lit:Literal, b4:Logic4)->
                    Or4(Bot,
                        b4))
                   (p(ci)) ...

=    {simplifying}

     ... False ...

=    {back to the top level}

   { ci | ci in? domain(p) &&
          Refines(False,
                  False)}

=    {simplifying}

     domain(p).
```

That is, *all* clauses are initially unsatisfied, as summarized in

```
theorem initialize_mkUnsatClauses is
   fa(p:CNF) mkUnsatClauses(p, mapFrom(varsOf(p), (fn var -> Bot)))
             = domain(p)
```

The other case to handle is when the state is changed via the `updateState` operation. In the following derivation, we are looking for a definition of the new value of `unsatCs'` in terms of `unsatCs`.

```
assume: unsatCs = mkUnsatClauses(st.prop,st.varVal),
        st' = updateState(st, v, val),
        st'.prop = st.prop,
        st'.varVal = (update st.varVal v val)
source: unsatCs' = mkUnsatClauses(st'.prop,st'.varVal)

=    {substituting st'.varVal}

   unsatCs' = mkUnsatClauses(st'.prop, updateState(st, v, val).varVal)

=    {use st'.prop = st.prop, and unfold mkUnsatClauses}

   {ci | ci in? domain(st.prop) &&
           Refines(Eval4Clause((ci),updateState (st, v, val).varVal),
                   False)}

=    {distributing Eval4Clause over updateState}

   {ci | ci in? domain(st.prop) &&
           Refines(Eval4Clause(p(ci),st.varVal) Or4
                   (case val of
                     | true  -> if v in? posLits(p(ci)) then True else Bot
                     | false -> if v in? negLits(p(ci)) then True else Bot,
                   False))}

=    {distributing Refines over Or4 in LHS}

   {ci | ci in? domain(st.prop) &&
         Refines(Eval4Clause(p(ci),st.varVal), False) &&
         Refines(case val of
                  | true  -> if v in? posLits(p(ci)) then True else Bot
                  | false -> if v in? negLits(p(ci)) then True else Bot,
                False)}

=    {using set law: {x|P(x) & Q(x)}  = {x|P(x)}\{x|~Q(x)}
```

```
   set_difference({ci | ci in? domain(st.prop) &&
                        Refines(Eval4Clause(p(ci),st.varVal), False)},
                  {ci | ci in? domain(st.prop) &&
                        ~Refines(case val of
                                  | true  -> if v in? posLits(p(ci))
                                             then True else Bot
                                  | false -> if v in? negLits(p(ci))
                                             then True else Bot,
                                 False)})

=    {folding definition of unsatCs}

   set_difference(unsatCs,
                  {ci | ci in? domain(st.prop) &&
                        ~Refines(case val of
                                  | true  -> if v in? posLits(p(ci))
                                             then True else Bot
                                  | false -> if v in? negLits(p(ci))
                                             then True else Bot,
                                 False)})

=    {using ~Refines(x,False) = (x=True) }

   set_difference(unsatCs,
                  {ci | ci in? domain(st.prop) &&
                        (case val of
                         | true  -> if v in? posLits(p(ci))
                                    then True else Bot
                         | false -> if v in? negLits(p(ci))
                                    then True else Bot) = True)})

=    {driving equality inward through the case statement and conditionals}

   set_difference(unsatCs,
                  {ci | ci in? domain(st.prop) &&
                        (case val of
                         | true  -> (if v in? posLits(p(ci))
                                    then True=True else Bot=True)
                         | false -> if v in? negLits(p(ci))
                                    then True=True else Bot=True))})

=    {simplifying}

   set_difference(unsatCs,
               {ci | ci in? domain(st.prop) &&
                     (case val of
                      | true  -> (if v in? posLits(p(ci))
                                  then true else false)
                      | false -> if v in? negLits(p(ci))
                                  then true else false))})
=   {simplifying}

   set_difference(unsatCs,
               {ci | ci in? domain(st.prop) &&
                     (case val of
                      | true  -> v in? posLits(p(ci))
                      | false -> v in? negLits(p(ci)))})

=   {introducing a top-level case analysis on val}

   case val of
    | true  -> set_difference(unsatCs,
               {ci | ci in? domain(st.prop) &&
                     (case val of
                      | true  -> v in? posLits(p(ci))
                      | false -> v in? negLits(p(ci)))})
    | false -> set_difference(unsatCs,
               {ci | ci in? domain(st.prop) &&
                     (case val of
                      | true  -> v in? posLits(p(ci))
                      | false -> v in? negLits(p(ci)))})

=   {simplifying}

   case val of
    | true  -> set_difference(unsatCs,
               {ci | ci in? domain(st.prop) &&
                     v in? posLits(p(ci))})
    | false -> set_difference(unsatCs,
               {ci | ci in? domain(st.prop) &&
                     v in? negLits(p(ci))})

=   {folding with varToClauses}

   case val of
    | true  -> set_difference(unsatCs, vtc(v).posClauses)
    | false -> set_difference(unsatCs, vtc(v).negClauses)
```

That is, the newly unsatisfied clauses are those in which variable `v` occurs positively when `val` is true, and those clauses in which variable `v` occurs negatively when `val` is false.

```
theorem distribute_mkUnsatClauses_over_update is
  fa(p:CNF, vm:Valuation4, vtc:VarToClauses,
     v:Variable, val:Boolean,
     cis:Set Nat, satcs:Map(Nat,Boolean))
     (vtc = computeVarToClauses(p,vm,cis,satcs)
      => (mkUnsatClauses(p, (update vm v (BooleantoLogic4(val))))
          = updateunsatClauses(mkUnsatClauses(p,vm), vtc, p, vm, v, val)))

op updateunsatClauses(usc:Set Nat, vtc: VarToClauses,
                      p:CNF, vm:Valuation4,
                      var:Variable, val:Boolean
                      | vtc = computeVarToClauses(p,vm,domain(p),
                                                  mkSatClauseMap(p,vm))
                        && usc = mkUnsatClauses(p,vm)): Set Nat =
   let pn:PosNeg = TMApply(vtc,var) in
   case val of
     | false -> set_difference(usc, pn.negClauses)
     | true  -> set_difference(usc, pn.posClauses)
```

The two theorems derived above are used to simplify the code after the type isomorphism transformation has been applied in `sat6`. In particular, the following part of the transformation script specifies that to construct `sat6`, first apply the type isomorphism transformation together with various simplifying theorems to `sat5`. Among those simplifying theorems are the two that we just calculated for enabling the incremental maintenance of `unsatCs`, specified to be applied left-to-right (lr).

```
sat6 = transform sat5 by {isomorphism(iso1, osi1)
                            (...
                             lr initialize_mkUnsatClauses,
                             ...
                             lr distribute_mkUnsatClauses_over_update,
                             ...)
                          ...}
```

2.5 Datatype Refinement

The derivation so far has been carried out in terms of relatively abstract datatypes, especially finite sets and maps. The last two steps of the CDB derivation are datatype refinement steps that translate the abstract types into compilable efficient representations. As with algorithm design, substitution is used as the refinement generation mechanism.

```
sat11 = sat10[DataTypes/MapsAsBTVectorsRef]

CDB = sat11[DataTypes/SetsAsListsRef]
```

The representation of maps is critical since access, incremental updates, and backtracking are frequent operations during search. Westfold developed an array-based implementation of maps that supports O(1) time access, O(1) time update, and O(1) time per level of backtrack. As MetaSlang has no arrays, the implementation is developed in CommonLisp but it is treated as a Specware morphism and applied by substitution.

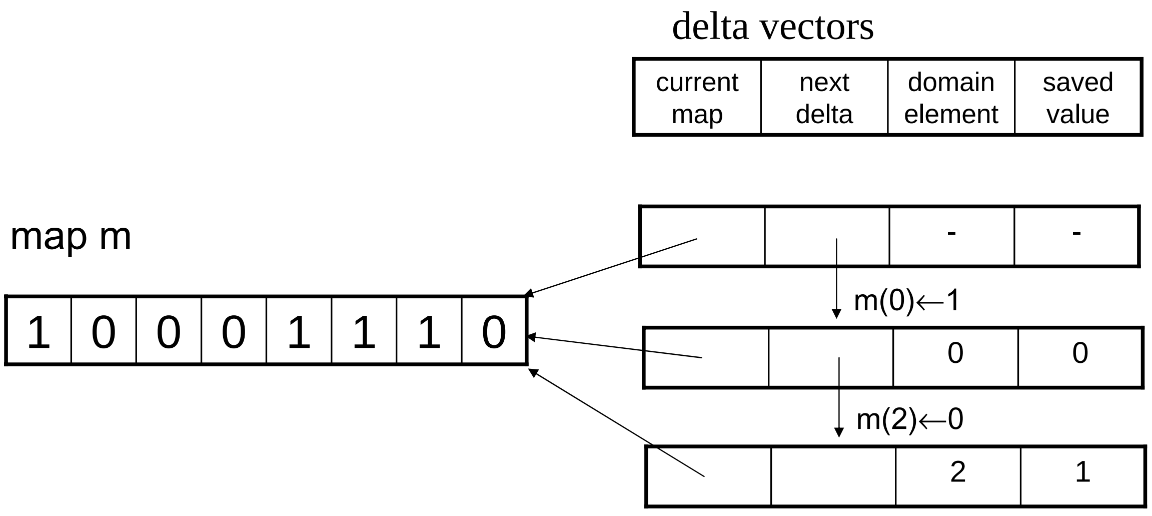

Figure 2: Array Representation of Maps with Version Metadata

Figure 2 shows the array representation of maps and their metadata in graphical form. Each delta vector represents a different version of the map. The actual reference to a version of the array is to a delta vector, whose first component points to the array. The "next delta" field is a pointer to the delta vector of the next version of the map. If the pointer is null, then the current value of the array is correct. Whenever an update is made to the map, then a new delta vector is generated, storing the original values of the domain and range elements that were just updated. When the program needs to access the map, via a delta vector, the access function first checks if the "next delta" pointer is null. If so, then the access is performed on the current state of the array. If not, then the chain of "next delta" pointers is followed and then the array elements are restored using the saved values in the delta vectors. In this way the array is restored and it can be safely accessed.

3 Performance

Over the course of this project we have generated a series of increasingly fast DPLL-like SAT algorithms. Figure 3 shows the improvements of over nine versions on a particular example that we have used as a benchmark (SAT/Benchmarks/aim-100-1-6-yes1-1.cnf). The vertical axis is $\log_{10}$ milliseconds of runtime. The figure shows a 6.1 orders-of-magnitude improvement in performance (1,370,000x speedup). The improvements are a mixture of improved algorithms, heuristics, optimizations, and data structures.

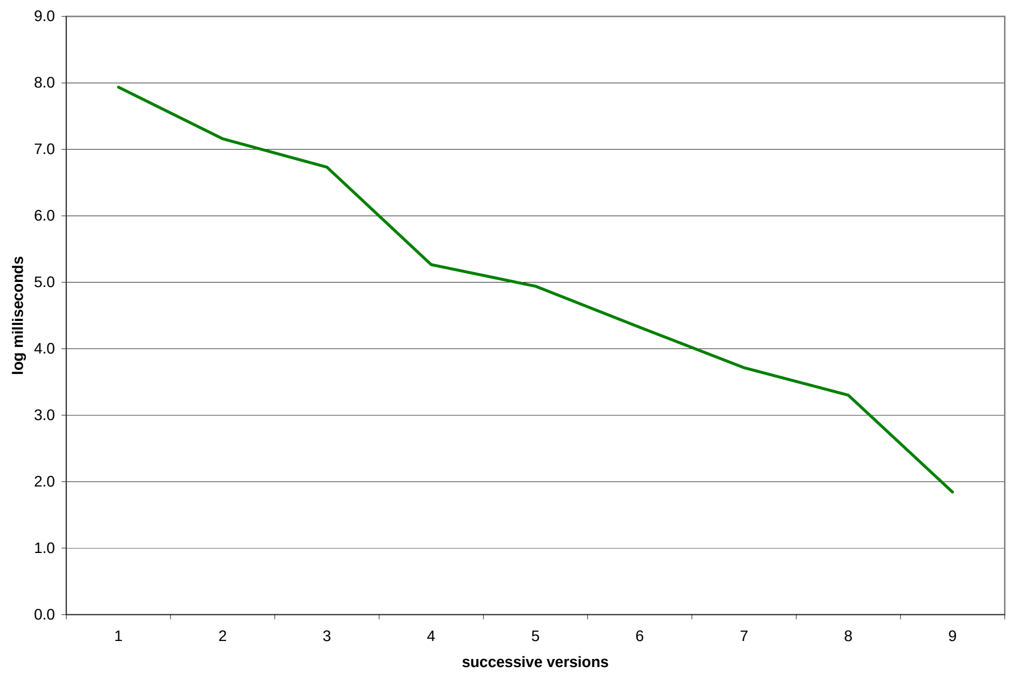

Figure 3: DPLL speedups during the project on DIMACS benchmark aim-100-1-6-yes1-1.cnf

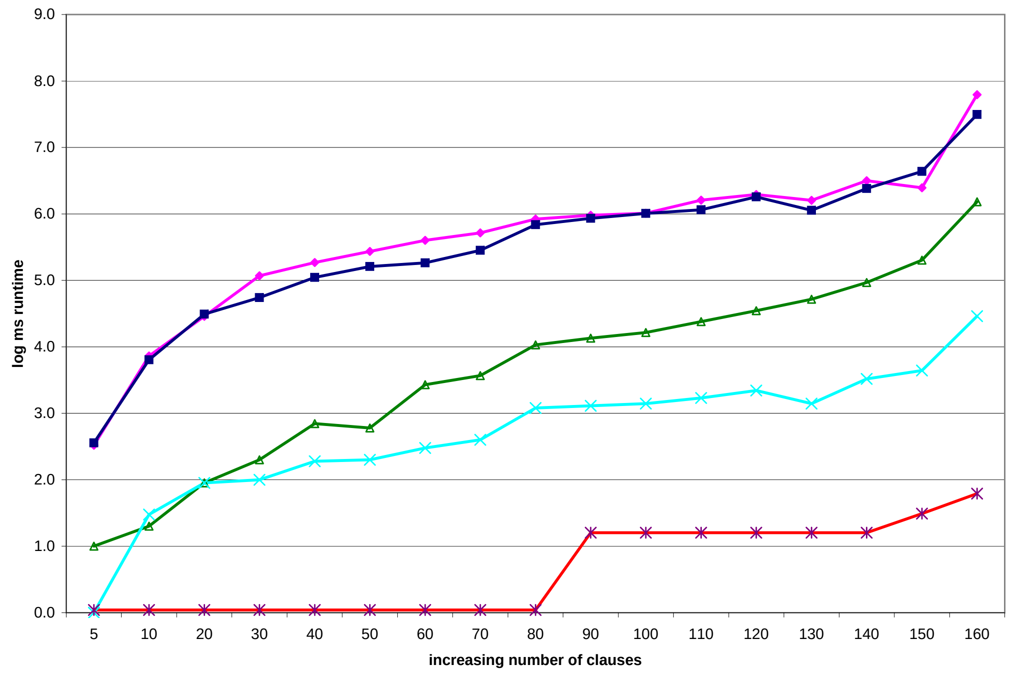

Figure 4: Ablation Study

The next figure shows an ablation study on our current best algorithm (CDBmom.sw). That is, we show the effect of removing design knowledge from the derivation process. The figure shows the performance effects of various stages in the refinement process in CDB.sw. The curves depict the runtimes of successive versions of the derivation (in $\log_{10}$ milliseconds on a IBM T43 laptop). The problem data is obtained from the same DIMACS benchmark, (SAT/Benchmarks/aim-100-1-6-yes1-1.cnf), which has 100 variables and 160 clauses. Each version of the algorithm was run with the first 10,20,30,...160 clauses for a total of 16 increasingly constrained problem instances.

1. *sat1* – The topmost curve (pink) is the raw DPLL algorithm with simple conflict-directed backjumping, with no finite differencing, and with default data structure implementations (sets and maps implemented as lists).

2. *sat4* – The second curve (blue) results from finite differencing of several set expressions, including the set of indices of clauses, the set of variables, and the sizes of these sets. The change does not appear dramatic, but the small visual difference is about a factor of two speedup.

3. *sat6* – The third curve (green) results from finite differencing to maintain a map that cross indexes variables and clauses: which open clauses does each variable occur in. There is a significant speedup (50-100x) here due to incrementally maintaining this cross-linking.

4. *sat10* – The fourth curve (light blue) results from finite differencing to maintain the count of open literals per clause. This brings about a significant speedup since it allows much faster access to unit clauses for propagation.

5. *CDB* – The fifth curve (red) results from refining the map datatypes to a specialized array representation that builds in backtracking information. The effect is to allow O(1) time backup per level of the search tree. The performance improvement is dramatic - about 310x for the full problem. The staircasing in the data is due to the fact that the laptop-based timer has a granularity of about 16ms.

4 New SAT Algorithm: Sequent Normal Form

We discovered what appears to be a new SAT algorithm, and it is not based on the DPLL foundation. The new algorithm is based on a sequent representation and a lattice-based view of valuations. The algorithm stems from Smith and Westfold's work on high-performance scheduling algorithms [15, 16]. A basic derivation of the algorithm is specified in SAT/Sequent.sw.

There are several key ideas in the approach, which can be explained in terms of Global Search theory. First, we use the lattice structure of Booleans (with implication as the partial order) rather than Kleene 4-logic. The initial space is a valuation in which all variable are *false*, rather than *unknown* as in DPLL. This *false*-valuation is the bottom element of the lattice of valuations. Note that most clauses will be satisfied by this initial valuation – all clauses except those that contain only positive variables are satisfied initially. Refinement of a space corresponds to moving some of the variable valuations upward (i.e. from `false` to `true`). Thus the initial space denotes all possible valuations by pointwise refinement of the variable valuations. More generally, a arbitrary space is represented by a valuation that denotes all possible refinements of the valuation.

Second, we treat clauses as *sequents* where we separate literals into positive and negative variables. We only need to refine positive variables in a sequent after all the negative variables have refined to `true`. The idea is to treat propagation as in definite constraint resolution. Intuitively, we should end up with many fewer resolutions/inferences, because the unit resolutions that DPLL would perform are handled intrinsically because of the initial assignment of *false* to all variables. Since DPLL spends most of its time performing unit resolution, we hope that this algorithm would finesse most of that time and obtain better performance.

The basic algorithm derived currently via `Sequent.sw` performs relatively few unit resolutions (compared to CDB), but it backtracks more since it doesn't yet have any form of conflict analysis and backjumping. Much more work needs to be done to explore various algorithmic aspects like conflict analysis, as well as the heuristics, optimizations, and data structures that are needed to fully development the algorithm's potential.

The derivation features various Finite Differencing steps that improve efficiency. Similar to CDB.sw, we maintain a cross-index map `varToSeqs` that is used to indicate which sequents are affected by a change to a given variable; i.e. if a variable's value is increased (from *false* to *true*) which other sequents may be triggered (because the variable occurs negatively in them). We also maintain a counter for the number of unsatisfied negative literals in a sequent `bodyCnt`, as in the Gallier-Downing algorithm for linear-time solving of Horn clauses. This allows fast detection of triggered sequents. It also allows linear-time solution of Horn-SAT instances. We also maintain which all-negative sequents are possibly violated `testSequents`, which definite/unit sequents are triggered `propagateSequents`, and which indefinite sequents are triggered for use in branching `branchSequents`.

The same datatype refinements are performed as in CDB.sw

5 Concluding Remarks: Future Work

The main general contribution of this project has been to lay out a standard pattern for the derivation of correct-by-construction SAT solvers. We have used Specware to demonstrate how to follow the pattern and mechanically generate several SAT solvers, including a new SAT algorithm. Given the time limits on the project, the generated codes incorporate some, but not all, of the features of the current best SAT solvers, so the currently generated solvers are fast, but require further development to match or beat the performance of state-of-the-art solvers.

The fact that the pattern suggested a new SAT algorithm is an encouraging sign. Again, given the resource limitations of the current project, further development of the SNF algorithm is needed to determine the conditions under which it is competitive with DPLL-based algorithms.

The derivations in the delivered SATware system provide automatic support for applying simplification and finite difference transformations, as well as automatic application of library-supplied datatype refinements and user-generated Global Search refinements. The key missing step in the derivations is lack of machine support for generating the Global Search refinement (e.g. GStoCDB.sw). In sections 2 and 3 above we lay out in detail the calculations that are needed to create this refinement. The older KIDS system [14] had a modified first-order prover that could generate refinements of this kind. We are currently studying how to adapt a higher-order prover, such as Isabelle or PVS, to perform the needed calculations.

Despite the six orders-of-magnitude speedup manifested by our development of the CDBmom algorithm, there are still dramatic improvements to be made. There are many more simplifications, finite differencing steps (e.g. watched literals [9]), and improved data structures that we could exploit. More importantly, there are several major algorithmic improvements that we did not have time to incorporate. It would have been easy to simply hack these into one of our algorithm schemes, but the point of the project was to perform systematic derivation of the algorithms from reusable design theories, so that others can derive variations.

1. *Optimal conflict analysis and backjumping* – We only began to to develop a fully general theory of conflict analysis, and there is only a basic version in the current CDBmom code. The best current theory is based on Unique Implication Points of an implication graph [12, 8, 18]. The implication graph represents the logical structure of decisions leading up to a failure, and unique implication points are dominators (or choke-points) of the failure.

   The question is how to lift the theory of conflict analysis to abstract Global Search theory, so that we can readily apply the concept to other problems. The general notion is that we want to infer the (logically) weakest explanation for the specific failure that occurs during search (i.e. when the pruning test fails). In SAT it is a clause that fails (evaluates to *false*). In other problems, the specific point of failure depends on the pruning test Phi. We want the weakest explanation for the failure, since its negation will be a strongest necessary condition of success; i.e. a strong propagation constraint. Our current hypothesis is this: each split in Global Search corresponds to a case analysis in which a disjunct is incorporated into the representation of a set. Each propagation step is an inference of a formula from the current subspace representation. The trick to generalizing conflict analysis to Global Search theory is to capture these logical correspondences in an implication graph and then to calculate a UIP-based set of formulas as a weakest sufficient reason for the current failure (of Phi).

2. *Learning* – Incorporating the negation of the conflict formula into the clause set has had tremendous effect in current SAT solvers. The challenge for abstract Global Search theory is to see how to incorporate the negated conflict formula into the code. For SAT, this is particularly easy since the negation of a conjunction of literals is a clause, by definition. For other problems, it is less clear how to incorporate the inferred clause into the code. For example, if a CSP problem fails and we obtain a clause from the negation of the inferred conflict, the clause may not be representable in the data structures available to the algorithm. This may be part of this reason that researchers find it convenient to reduce problems to SAT. However, we believe that it is likely that there are general principles for representing Global Search spaces such that formulas gleaned from failures can be readily exploited to improve further search. It would be fruitful to explore such general principles rather than continue to focus on methods for reducing problems to SAT that lose structure.

3. *Preprocessing* – Preprocessing is commonly performed to simplify the input formula and precompute/cache as much information as possible to preclude the need for redundant computation later. Many of these steps can be performed by Finite Differencing. In our current algorithms, the only preprocessing that is performed is propagation via Unit Rule and Pure Literal Rule, and to create Finite Differencing variables that speed up later search.

4. *Restarts* – Restarting the search after a certain number of failures has been shown to be effective in dealing with the heavy-tail phenomenon of search [5] while retaining completeness. Adding this feature to the Global Search scheme would be simple, but it leaves open some parameters whose choice tends to be heuristic and empirical.

References

[1] Coglio, A. Transformation refinement for specware. Tech. rep., Kestrel Institute, 2007.

[2] Cousot, P., and Cousot, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL* (1977), ACM, pp. 238–252.

[3] Davis, M., Logemann, G., and Loveland, D. A machine program for theorem-proving. *Commun. ACM 5*, 7 (1962), 394–397.

[4] Davis, M., and Putnam, H. A computing procedure for quantification theory. *J. ACM 7*, 3 (1960), 201–215.

[5] Gomes, C. P., Selman, B., and Kautz, H. Boosting combinatorial search through randomization. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI'98)* (Madison, Wisconsin, 1998), pp. 431–437.

[6] Manna, Z., and Waldinger, R. Special relations in automated deduction. *Journal of the ACM 33*, 1 (January 1986), 1–59.

[7] Marek, V. W. *Mathematics of Satisfiability.* 2005.

[8] Marques-Silva, J., and Sakallah, K. Grasp: a search algorithm for propositional satisfiability. *IEEE Transactions on Computers 48*, 5 (1999), 506 – 521.

[9] Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., and Malik, S. Chaff: engineering an efficient sat solver. In *DAC '01: Proceedings of the 38th conference on Design automation* (2001), ACM Press, pp. 530–535.

[10] Paige, R., and Koenig, S. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems 4*, 3 (July 1982), 402–454.

[11] Purdom, P., and Haven, G. N. Probe order backtracking. *SIAM Journal on Computing 26*, 2 (1997), 456–483.

[12] Silva, J., and Sakallah, K. Conflict analysis in search algorithms for propositional satisfiability, 1996.

[13] Smith, D. R. Structure and design of global search algorithms. Tech. Rep. KES.U.87.12, Kestrel Institute, November 1987.

[14] Smith, D. R. KIDS – a semi-automatic program development system. *IEEE Transactions on Software Engineering Special Issue on Formal Methods in Software Engineering 16*, 9 (1990), 1024–1043.

[15] Smith, D. R., Parra, E. A., and Westfold, S. J. Synthesis of planning and scheduling software. In *Advanced Planning Technology* (1996), A. Tate, Ed., AAAI Press, Menlo Park, pp. 226–234.

[16] Westfold, S., and Smith, D. Synthesis of efficient constraint satisfaction programs. *Knowledge Engineering Review 16*, 1 (2001), 69–84. (Special Issue on AI and OR).

[17] Zhang, L. Solving QBF with combined conjunctive and disjunctive normal form. In *Twenty-First National Conference on Artificial Intelligence (AAAI 2006)* (2006).

[18] Zhang, L., Madigan, C. F., Moskewicz, M. W., and Malik, S. Efficient conflict driven learning in boolean satisfiability solver. In *ICCAD* (2001), pp. 279–285.

A Lattice-Based Laws

A variety of laws related to lattices arise naturally and commonly in derivations. The reason is that many domain and types have lattice structure, and there is a fairly rich theory about lattices. Rather than replicate instances of the same lattice theorems/laws for each type, we state them in their abstract lattice form. To give some indication of their power, we also list some common instances.

A.1 Lattices

A lattice $\langle L, \sqcap, \sqcup, \leq \rangle$ is a partial order $\langle L, \leq \rangle$ together with a least upper bound operator $\sqcup$, called *join*, a greatest lower bound operator $\sqcap$, called *meet*.

A bounded lattice $\langle L, \sqcap, \sqcup, \bot, \top, \leq \rangle$ has a universal lower bound $\bot \leq x$ for all $x \in L$, and a universal upper bound $x \leq \top$ for all $x \in L$. A lattice is *complete* if every subset of L has a least upper bound in L and a greatest lower bound in L.

A.2 Lattice Quantifiers

A lattice quantifier is the reduction of join or meet over a set of lattice elements. It is convenient to define notation to cover common cases of indexing over sets. The general case is expressed:

$$\bigsqcup_{a|P(a)} f(a)$$

and dually

$$\bigsqcap_{a|P(a)} f(a)$$

which quantify over the set $\{a \mid a \in L \ \wedge \ P(a)\}$.

A.3 Quantifier Elimination Laws

Lattice quantifiers are useful in formulating problems, and laws to eliminate them are a powerful tool during calculation. Listed below are elimination laws for lattice-based quantifiers in which we have functions from a preorder $\langle A, \preceq \rangle$ to a lattice $\langle L, \sqcap, \sqcup, \leq \rangle$.

Monotone $F : \langle A, \preceq \rangle \to \langle L, \sqcap, \sqcup, \leq \rangle$			
1.1	$\bigsqcup_{a \preceq \hat{a}} F(a) = F(\hat{a})$	$\bigsqcap_{\check{a} \preceq a} F(a) = F(\check{a})$	1.2

Antimonotone $F : \langle A, \preceq \rangle \to \langle L, \sqcap, \sqcup, \leq \rangle$			
-1.1	$\bigsqcup_{\check{a} \preceq a} F(a) = F(\check{a})$	$\bigsqcap_{a \preceq \hat{a}} F(a) = F(\hat{a})$	-1.2

Specialization to Predicates

Monotone $F : \langle A, \preceq \rangle \to \langle \mathit{Boolean}, \wedge, \vee, \Rightarrow \rangle$			
2.1	$\bigvee_{a:A \mid a \preceq \hat{a}} F(a) = F(\hat{a})$	$\bigwedge_{a:A \mid \check{a} \preceq a} F(a) = F(\check{a})$	2.2
2.1	$\exists(a : A \mid a \preceq \hat{a}) F(a) = F(\hat{a})$	$\forall(a : A \mid \check{a} \preceq a) F(a) = F(\check{a})$	2.2

Antimonotone $F : \langle A, \preceq \rangle \to \langle \mathit{Boolean}, \wedge, \vee, \Rightarrow \rangle$			
-2.1	$\exists(a : A)\ (\check{a} \preceq a \wedge F(a)) = F(\check{a})$	$\forall(a : A)\ (a \preceq \hat{a} \Rightarrow F(a)) = F(\hat{a})$	-2.2

Specialization to Propositional Formulas

Monotone $F : \langle \mathit{Boolean}, \Rightarrow \rangle \to \langle \mathit{Boolean}, \wedge, \vee, \Rightarrow \rangle$			
3.1	$\exists(a : \mathit{Boolean})\ F(a) = F(\mathit{true})$	$\forall(a : \mathit{Boolean})\ F(a) = F(\mathit{false})$	3.2

Antimonotone $F : \langle \mathit{Boolean}, \Rightarrow \rangle \to \langle \mathit{Boolean}, \wedge, \vee, \Rightarrow \rangle$			
-3.1	$\exists(a : \mathit{Boolean})\ F(a) = F(\mathit{false})$	$\forall(a : \mathit{Boolean})\ F(a) = F(\mathit{true})$	-3.2

A.4 Quantifier Change Laws

Listed below are quantifier change laws. It often happens that we have an expression that is quantified over one set S, but for purposes of calculation, we need it quantified over a different set T. The laws presented below show how to effect such a change.

Listed below are quantifier change laws with respect to a lattice $\langle L, \sqcup, \sqcap, \leq \rangle$.

$$\boxed{\frac{S \subseteq T \text{ and } g : T \to L}{\bigsqcup_{x \in S}\ g(x)\ \leq\ \bigsqcup_{x \in T}\ g(x)}} \qquad \boxed{\frac{S \subseteq T \text{ and } g : T \to L}{\bigsqcap_{x \in S}\ g(x)\ \geq\ \bigsqcap_{x \in T}\ g(x)}}$$

or, more generally,

$$\boxed{\frac{h : S \to T \text{ and } g : T \to L}{\bigsqcup_{x \in S}\ g(h(x))\ \leq\ \bigsqcup_{x \in T}\ g(x)}} \qquad \boxed{\frac{h : S \to T \text{ and } g : T \to L}{\bigsqcap_{x \in S}\ g(h(x))\ \geq\ \bigsqcap_{x \in T}\ g(x)}}$$

Special Cases:

1. L is the *Boolean* lattice: $\langle Boolean, \wedge, \vee, \Rightarrow \rangle$

$$\boxed{\frac{S \subseteq T \text{ and } g : T \to Boolean}{\bigvee_{x \in S}\ g(x)\ \Longrightarrow\ \bigvee_{x \in T}\ g(x)}} \qquad \boxed{\frac{S \subseteq T \text{ and } g : T \to Boolean}{\bigwedge_{x \in S}\ g(x)\ \Longleftarrow\ \bigwedge_{x \in T}\ g(x)}}$$

2. L is the lattice of (polymorphic) finite sets: $\langle Set(\alpha), \cup, \cap, \subseteq \rangle$

$$\boxed{\frac{S \subseteq T \text{ and } g : T \to L}{\bigcup_{x \in S}\ g(x)\ \subseteq\ \bigcup_{x \in T}\ g(x)}} \qquad \boxed{\frac{S \subseteq T \text{ and } g : T \to L}{\bigcap_{x \in S}\ g(x)\ \supseteq\ \bigcap_{x \in T}\ g(x)}}$$

4. L is the lattice of Integers: $\langle Integer, max, min, \leq \rangle$

$$\boxed{\frac{S \subseteq T \text{ and } g : T \to L}{\max_{x \in S}\ g(x)\ \leq\ \max_{x \in T}\ g(x)}} \qquad \boxed{\frac{S \subseteq T \text{ and } g : T \to L}{\min_{x \in S}\ g(x)\ \geq\ \min_{x \in T}\ g(x)}}$$

A.5 Inference Rules

Polarity

$$\boxed{\dfrac{\text{Isotone } F : \langle A, \preceq \rangle \to \langle L, \leq \rangle}{a \preceq b \implies F(a) \leq F(b)}} \qquad \boxed{\dfrac{\text{Antitone } F : \langle A, \preceq \rangle \to \langle L, \leq \rangle}{a \preceq b \implies F(a) \geq F(b)}}$$

Resolution

$$\frac{\text{Isotone } E : \mathit{Boolean} \to \mathit{Boolean}, \text{ Antitone } F : \mathit{Boolean} \to \mathit{Boolean}}{E(a) \wedge F(a) \implies E(\mathit{false}) \vee F(\mathit{true})}$$

$$\frac{\text{Isotone } E : \mathit{Boolean} \to \mathit{Boolean}, \text{ Antitone } F : \mathit{Boolean} \to \mathit{Boolean}}{E(a) \vee F(a) \impliedby E(\mathit{true}) \wedge F(\mathit{false})}$$

A Consistent Refinement: Generalizing the Pure Literal Rule

Many problems are specified in such a way that (1) there may exist many solutions and (2) they are sparsely distributed. It would be desirable to have techniques that that narrow down the solution space while preserving the existence of solutions (or preserving satisfiability). More precisely, in a *consistent refinement* of problem P to problem P', all solutions to P' are solutions to P, and if P has a solution then so does P'. With skillful application and some luck, we can define a restricted search space that has better structure and/or is denser with solutions.

For problems that can be viewed as finding bounds over a lattice (such as conventional optimization problems), this note presents a general technique for consistent refinement. The pure literal rule from SAT [7] falls out as a special case.

PROPOSITION A.1 *Let* $F : \langle A, \preceq \rangle \rightarrow \langle L, \sqcap, \sqcup, \leq \rangle$ *be a function from a preorder to a lattice.*

If F *is monotone,*
then (1.1) $\sqcup_{a \preceq \hat{a}} F(a) \;=\; F(\hat{a})$
and (1.2) $\sqcap_{\check{a} \preceq a} F(a) \;=\; F(\check{a})$.

If F *is antimonotone,*
then (-1.1) $\sqcap_{\check{a} \preceq a} F(a) \;=\; F(\check{a})$
and (-1.2) $\sqcup_{a \preceq \hat{a}} F(a) \;=\; F(\hat{a})$.

Proof: (1.1) For monotone F and some arbitrary value $a \preceq \hat{a}$, we have $F(a) \leq F(\hat{a})$, or, equivalently, $F(a) \sqcup F(\hat{a}) = F(\hat{a})$. We then have

$$\begin{aligned} \sqcup_{a \preceq \hat{a}} F(a) &= \sqcup_{a \preceq \hat{a}} F(a) \sqcup F(\hat{a}) \\ &= \sqcup_{a \preceq \hat{a}} F(\hat{a}) \\ &= F(\hat{a}). \end{aligned}$$

The case (1.2) for $\sqcap$ is dual, and the cases (-1.1, -1.2) when F is antimonotone are analogous. $\square$

Proposition A.1 and various specializations are summarized in tabular form in Appendix A.

Notes

- The proof does not depend on how F is defined. It could be discrete or continuous. It could be computable or not.
- The effect of Proposition A.1 is to reduce the dimensionality of a problem, almost for free in many cases, thereby reducing the computational complexity of problem-solving. It can be applied statically to reformulate a problem specification, or dynamically to refine a problem instance (as in SAT).

The pure literal rule in SAT is a special case of Proposition A.1.

Corollary A.1 *Interpret F as a propositional formula according to*

$$\begin{array}{rcl} \langle A, \preceq \rangle & \mapsto & \langle \mathit{Boolean}, \Rightarrow \rangle \\ \langle L, \sqcap, \sqcup, \leq \rangle & \mapsto & \langle \mathit{Boolean}, \wedge, \vee, \Rightarrow \rangle \\ \check{a} & \mapsto & \mathit{false} \\ \hat{a} & \mapsto & \mathit{true} \end{array}$$

where

$$\sqcup_{a \Rightarrow true} F(a)$$

is conventionally expressed as

$$\exists (a : \mathit{Boolean})\ ((a \Rightarrow \mathit{true})\ \wedge\ F(a))$$

or simply

$$\exists (a)\ F(a);$$

similar comments apply to $\sqcap$.

If F *is monotone,*
then (3.1) $\exists(a)\ F(a) \ = \ F(\mathit{true})$
and (3.2) $\forall(a)\ F(a) \ = \ F(\mathit{false})$.

If F *is antimonotone,*
then (-3.1) $\exists(a)F(a) \ = \ F(\mathit{false})$
and (-3.2) $\forall(a)F(a) \ = \ F(\mathit{true})$.

It is easy to show that a CNF formula $F(a)$ is monotone in a iff a is a pure positive literal (i.e. all occurrences of a occur positively), and dually for the antimonotone/negative case. Cases (3.1) and (-3.1) are the basis for the pure literal rule in SAT.

Notes

- *Applications* – Corollary A.1 reveals that the SAT-specific concept of "pure literal" can be seen as a special case of a more general concept - a monotone parameter to a function. This more general understanding allows us to derive analogues of the pure-literal rule in other SAT formats (e.g. DNF, XOR form, negation normal-form, etc.) and to other problems, such as QBF and general CSP problems. Monotonicity analysis is the key.
- *Quantifier Elimination* – The Corollary is a special form of quantifier elimination. Can quantifier elimination ideas be used to further generalize this result? Can familiar quantifier elimination mechanisms such as Fourier-Motzkin bw usefully generalized to lattices?
- *Consistency of Formal Specifications* – A specification is a collection of type symbols and function symbols, together with axioms to constrain the semantics of the vocabulary. When a specification is used to construct a correct program, we are essentially asking for a witness to the satisfiability or consistency of the specification. One of the hazards of

a refinement process is refining to an inconsistent specification (which admits no implementation/model). Techniques for assuring that a specification is consistent would aid in keeping a refinement process on track. Consistency of a specification

$$S = \langle \{t_1, t_2, \cdots, t_m\},\ \{f_1, f_2, \cdots, f_n\},\ \mathit{Axioms} \rangle$$

where t_i are types and f_i are functions, is the problem of deciding

$$\exists (t_1, t_2, \cdots, t_m)\, \exists (f_1, f_2, \cdots, f_n)\ \mathit{Axioms}(t_1, t_2, \cdots, t_m,\ f_1, f_2, \cdots, f_n).$$

In this context, Corollary A.1 may provide some help in constructing a (perhaps artificial) model of the axioms to demonstrate consistency.

A.1 Monotonicity Analysis

Proposition A.1 relies on monotonicity analysis for its application. Logically, the proposition holds for functions in general, regardless of how they are defined. Practically however, its application depends on determining monotonicity of arguments which, in turn, depends on analyzing how the function is defined.

A function is analyzed for monotonicity of its arguments by recursion on the structure of its definition and using the monotonicity properties of the terms (basic functions and combinators) used in the definition. As discussed below, for propositional and first-order recursive functions, analyzing the monotonicity properties of a term can be performed in linear time. The key idea of the analysis is abstract interpretation [2] in which we map a function definition into the abstract domain of polarity algebra (cf. [6]).

A.1.1 Polarity Algebra

The goal is to decide if function $F : A \to C$ is (anti)monotone in A. Generally, F can take multiple arguments, but for the purposes of monotonicity analysis we regard the other arguments/parameters as fixed. We analyze the definition of F by mapping its term structure into a term in Polarity algebra, which is a four-element lattice with additional structure:

$$\begin{tikzcd}
& \pm \arrow[dl, dash] \arrow[dr, dash] & \\
+ \arrow[dr, dash] & & - \arrow[dl, dash] \\
& \bot &
\end{tikzcd}$$

The intention is to label each subterm of the definition with an element of the lattice. In particular, for F:

A label of $\perp$ means that the monotonicity of F is unknown;
A label of $+$ means that F is monotone ($a \leq b \Rightarrow F(a) \leq F(b)$);
A label of $-$ means that F is antimonotone ($a \leq b \Rightarrow F(a) \geq F(b)$);
A label of $\pm$ means that F is substitutive ($a = b \Rightarrow F(a) = F(b)$)

Formally, Polarity algebra has structure similar to a Boolean algebra

$$\langle \{\perp, +, -, \pm\}, \sqsubseteq, \sqcap, \sqcup, \complement, \mathrm{I} \rangle$$

where $\sqsubseteq$ is a partial order, $\sqcap$ is the meet of the lattice, and $\sqcup$ is the join, with the usual lattice semantics shown in the table:

p	q	$p \sqsubseteq q$	$p \sqcap q$	$p \sqcup q$
$\perp$	$\perp$	*true*	$\perp$	$\perp$
$\perp$	$-$	*true*	$\perp$	$-$
$\perp$	$+$	*true*	$\perp$	$+$
$\perp$	$\pm$	*true*	$\perp$	$\pm$
$-$	$\perp$	*false*	$\perp$	$-$
$-$	$-$	*true*	$-$	$-$
$-$	$+$	*false*	$\perp$	$\pm$
$-$	$\pm$	*true*	$-$	$\pm$
$+$	$\perp$	*false*	$\perp$	$+$
$+$	$-$	*false*	$\perp$	$\pm$
$+$	$+$	*true*	$+$	$+$
$+$	$\pm$	*true*	$+$	$\pm$
$\pm$	$\perp$	*false*	$\perp$	$\pm$
$\pm$	$-$	*false*	$-$	$\pm$
$\pm$	$+$	*false*	$+$	$\pm$
$\pm$	$\pm$	*true*	$\pm$	$\pm$

Polarity algebra also has an identity operator I and a complementation operator $\complement$ defined by

p	$\mathrm{I}\ p$	$\complement p$
$\perp$	$\perp$	$\perp$
$-$	$-$	$+$
$+$	$+$	$-$
$\pm$	$\pm$	$\pm$

Note that the complementation operator differs slightly from that in a Boolean Algebra, in that it preserves rather than interchanging $\pm$ and $\perp$.

A.1.2 Analyzing a Propositional Function

A propositional function/formula is built out of variables (proposition letters), plus the typical propositional connectives: conjunction ($\wedge$), disjunction ($\vee$), and negation ($\neg$). Conjunction and disjunction are monotone in both of their arguments, and negation is antimonotone in its argument. Other propositional connectives can be similarly analyzed.

Suppose that the definition of F has the form $F(a) = t[a]$ where t is a term built out of the propositional connectives plus a plus other variables.

An abstraction function $abs : PropositionTerm \rightarrow PolarityTerm$ is defined as follows:

$$\begin{array}{rcl} abs(v) & = & \bot \quad if\ v \neq a \\ abs(a) & = & + \\ abs(g(t_1, \ldots, t_n)) & = & pol(g)(abs(t_1), \ldots, abs(t_n)) \end{array}$$

where *pol* maps base functions to polarity functions that reflect their monotonicity properties:

$$\begin{array}{rcl} pol(\wedge) & = & \sqcap \\ pol(\vee) & = & \sqcap \\ pol(\neg) & = & \mathsf{C} \end{array}$$

Defined connectives can be mapped similarly:

$$\begin{array}{rcl} pol(\Rightarrow) & = & \lambda(p, q)(\mathsf{C}(p)\ \sqcap\ q) \\ pol(\ \equiv\ ) & = & \lambda(p, q)(\pm(p)\ \sqcap\ \pm(q)) \end{array}$$

For example, the propositional function

$$F(a, b, c)\ =\ a\ \wedge\ (b \Rightarrow c)\ \wedge\ \neg c$$

abstracts to the polarity function (in a)

$$p_F(a, b, c)\ =\ +\ \sqcap\ (\mathsf{C}(\bot) \sqcap\ \bot)\ \sqcap\ \mathsf{C}(\bot)$$

which simplifies to

$$p_F(a, b, c)\ =\ +$$

Similarly, we calculate that F has polarity $-$ in b and $\pm$ in c.

A.1.3 Analyzing a Recursive First-Order Function

The propositional connectives are extended with functions, predicates, and first-order quantification. Assume that we just have if-then-else, function composition, and recursion as control constructs. Assume also that we map each primitive function used in our definition to a polarity function that reflects its monotonicity properties. We abstract propositional connectives as above. The control constructs are abstracted as follows.

1. Abstract a function composition
$$f(g(x))$$
to the composition of their polarity functions:
$$pol(f) \circ pol(g)$$
(using $\circ$ for function composition)
2. Abstract a conditional
$$\textsf{if } p \textsf{ then } t \textsf{ else } e$$
to
$$pol(t) \ \vee \ pol(e)$$
i.e. ignore the test and take the meet of the polarities of the then and else branches.
3. Abstract a function definition
$$f(x) \ = \ def$$
to an equation
$$v_f \ = \ pol(def)$$
between a polarity variable p_f and the abstracted polarity expression for *def*.

The result is a set of recurrence equations over the variables denoting the polarities of defined functions. Solving for the least fixpoint yields the polarity labeling for each defined function.

Example: Define

```
len(x) = if empty(x) then 0 else succ(len(cdr(x)))
```

and suppose we want to analyze for polarity/monotonicity of *len* with respect to the order $l1 \leq l2$ if $l1$ is a prefix of $l2$. The following recurrence gets set up via the abstraction process:

$v_{len} \ = \ \perp \sqcap \mathrm{I} \ (v_{len})$

whose least fixpoint is $p_{len} = +$, as expected.

Remark 1: Since the Polarity lattice has height three (the length of the longest ascending chain), there are at most three iterations of the recurrence that need to performed in order to reach a least fixpoint. Consequently, polarity analysis of a recursive first-order definition can be found in time that is linear in the size of the defining terms.

Remark 2: The abstraction of primitive functions to polarity functions must be sound in order for the analysis to have meaning.

Remark 3: The polarity analysis of a function can give different results for different definitions. If the primitive functions are abstracted soundly, then the inferred analysis of the defined functions will also be sound. However, they may be weaker (higher in the lattice) than possible. For example, a monotone function may be analyzed to be substitutive. Both results are correct, but the stronger one is more useful.

A.2 Summary

These results show how to lift the essence of the pure-literal rule to a more abstract level. The abstract characterization then allows us to apply it to other SAT formulations and to other problems, such as QBF and general CSP problems. The generality and power of the technique requires tool support for monotonicity analysis.