Toward the Synthesis of Constraint Solvers

Douglas R. Smith
Stephen J. Westfold
Kestrel Institute
Palo Alto, CA 94304
{smith,westfold}@kestrel.edu
2 November 2013

Abstract

We develop a basic mathematical framework for specifying and formally designing high-performance constraint solving algorithms. The framework is based on concepts from abstract interpretation which generalizes earlier work on a Galois Connection-based model of Global Search algorithms. The main focus is on how to use the framework to automate the calculations necessary to construct a correct, high-performance solver. We present here the foundations for generating customized/native solvers for specified constraint satisfaction problems. Our thesis is that a native solver can always be generated for a constraint problem that outperforms a reduction to an existing solver.

1 Introduction

We take a deep look at constraint-solving algorithms from the point of view of synthesis technology. We explore the concepts and techniques that would allow us to automate the calculations necessary to construct a correct, high-performance solver. We present a basic mathematical framework for specifying and formally designing high-performance constraint solving algorithms. The framework is based on concepts from abstract interpretation which generalizes our previous work on a Galois Connection-based model of Global Search algorithms. The main focus is how to use the framework to automate the calculations necessary to construct a correct, high-performance solver. We lay the foundations for an automated synthesis system that can generate a customized, native solver for a specified constraint-based problem.

It is common practice in the constraint-solving community to solve a new problem P by building a reduction to a well-studied problem Q that has a well-engineered solver. For example, planning and scheduling problems can be reduced to SAT problems and solved on a SAT solver. In other cases, a scheduling problem might be reduced to a linear program and solved using a Simplex algorithm. One problem with this approach is that the reduction of P to Q often loses some key structure which cannot then be exploited by the Q-solver. Our thesis is that a native solver can always be generated for a constraint problem that outperforms a reduction to an existing solver (e.g. for SAT, SMT, Linear Programming, finite domain). Our larger goal then is to demonstrate the technology to enable the synthesis of customized algorithms for a given problem P that (1) incorporates all the best current algorithmic and data structuring technology, and (2) outperforms a P-solver that invokes a Q-solver.

Constraint Satisfaction/Optimization Problems (CSP/COP) provide a framework for formulating problems in terms of variables and constraints over the values that the variables can take on. A solution to a CSP is an assignment of values to the variables that satisfy all given constraints. Although there are important classes of CSP's that admit efficient algorithmic solutions (see Section 2.7), most interesting CSP's are NP-hard, requiring search.

Modern understanding of constraint-solving is that search takes place on two levels

1. theory-level inference – use the constraints to infer extensions to the current partial model, and to deductively infer new consequences of the constraints
2. model-level construction - extend or modify a partial assignment aiming to construct a feasible solution

In this view there are two extreme strategies:

1. pure theory inference – e.g. theorem provers/MC, static analysis
2. pure model search – e.g. simple backtracking, game-tree search

In general though, current best practice is based on alternating between theory search and model search, using models to guide the inference, and using inference to constrain model search [8, 2].

Our goal in this paper is to lay out a standard pattern for the derivation of correct-by-construction CSP solvers using the conflict-directed backjumping and learning paradigm. Our emphasis is on formalizing a design theory for this class of solvers, which is necessarily independent of the particulars of the constraint logic, and developing the calculations necessary to apply the design theory to specified problems.

Our criteria for developing a mathematical framework, and formalizing design knowledge within the framework are

1. the abstraction covers a broad range of useful problems,
2. allows simple calculations to generate best-practice code,
3. supports a direct generation of proofs of correctness for instances (for a formal assurance case).

These criteria are often in tension, and the resolution of that tension is part of the art of formalizing design knowledge.

The main results of this paper can be summarized as follows.

1. *Algorithm theory* – We developed and proved an algorithm theory for constraint solving with propagation, conflict detection and analysis, backjumping, and learning that is parametric on the constraint logic.

2. *Design Method for Constraint Propagation* – We proved that Arc Consistency is a best-possible constraint propagation mechanism for arbitrary CSPs, and then showed how to calculate optimal code for propagation. From Arc Consistency formula schemes we calculate simple Definite Constraints that can be instantiated into the optimal Definite Constraint Solver scheme.

3. *Theory of Conflict Analysis* – We explored several mathematical formalisms for generalizing conflict analysis to arbitrary logics. We discovered a general pattern for calculating resolution rules in a given logic, and proved how resolution can be iterated to soundly infer a new constraint for backjumping and learning purposes.

4. *Logic-independent Charactization of Constraint Resolution* – We explored several mathematical formalisms for generalizing conflict analysis to arbitrary logics. We discovered a general pattern for calculating resolution rules in a given logic, and proved how resolution can be iterated to soundly infer a new constraint for backjumping and learning purposes.

2 Theory of Constraint Solvers

2.1 Constraint Satisfaction Problems

The task of a Constraint Solver 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`.

A *Constraint Satisfaction Problem* (CSP) is classically specified by the following components. Given

1. Variables – a finite set of variables V
2. Domains – for each variable $v_i \in V$, a domain D_i of possible values for v_i
3. Constraints - predicates that constrain the joint values that variables can take on
4. Objective Function (optional) - to express preferences over solutions.

Find: an assignment of values to variables; i.e. a map $V \rightarrow \Pi D_i$ that satisfies all the constraints

We present several examples by giving an informal interpretation of CSP components:

$Problem$	$\mapsto$	k_Queens
$Input$	$\mapsto$	$k \geq 1$
$Variables$	$\mapsto$	$\{column_1, .., column_k\}$
$Domains$	$\mapsto$	$\{1, .., k\}$ for each $column_i$
$Constraints$	$\mapsto$	no two queens in the same row, column, or diagonal

Problem	$\mapsto$	*SAT*
Input	$\mapsto$	clauses $c_1, ...c_k$
Variables	$\mapsto$	proposition letters that occur in the constraints
Domains	$\mapsto$	$\{true, false\}$
Constraints	$\mapsto$	each constraint is a clause (disjunction of literals)

Problem	$\mapsto$	Linear Pseudo-Boolean
Input	$\mapsto$	integers $a_1, ...a_k, b_1, ..., b_k$
Variables	$\mapsto$	binary variables$\{x_1, ...x_k\}$
Domains	$\mapsto$	$\{0, 1\}$
Constraints	$\mapsto$	$\Sigma_{i=1}^{k} a_i \cdot x_i \;\leq\; b_i$ for integers $a_1, ...a_k, b_1, ..., b_k$

Problem	$\mapsto$	Scheduling
Input	$\mapsto$	tasks $tsk_1, ...tsk_k$ $resources r_1, ..., r_l$
Variables	$\mapsto$	*tasks*
Domains	$\mapsto$	$Resources \times Time$
Constraints	$\mapsto$	temporal constraints, resource capacity constraints, task/resource suitability,...

Problem	$\mapsto$	Program Analysis: Static Generation of Program Invariants
Variables	$\mapsto$	code locations $\{\ell_1, ...\ell_k\}$ of program P
Domains	$\mapsto$	State Information
Constraints	$\mapsto$	relations between code locations that reflect instruction semantics

Problem	$\mapsto$	Constructing a Specification Morphism in Program Synthesis
Variables	$\mapsto$	symbols of an algorithm/design theory
Domains	$\mapsto$	expressions of the specification language
Constraints	$\mapsto$	axioms of the algorithm theory

2.2 Specifying Constraint Problems

A specification of a constraint satisfaction problem (CSP) depends on a theory of constraints, which builds up the vocabulary for specifying CSP problems and for reasoning about them. In this section we give highlights of a CSP domain theory. Various CSP's are mainly differentiated by their *logic*; that is, by a satisfaction relation between a language for constraint expressions and a semantic domain. Below we introduce each component of a CSP in subsections.

2.3 CSP Semantics

In CSP's the semantics domain is modeled by finite maps from a given domain of variables and domain(s) of values $Valuation = \texttt{map}(\texttt{Var}, \texttt{Val})$. Furthermore, we will be interested in sets of such maps $Valuations = \texttt{Set}(\texttt{map}(\texttt{Var}, \texttt{Val}))$. In the following, we give a specification of CSP concepts expressed in the Metaslang language of Kestrel's Specware system [4].

```
type Variable
type Value
type Valuation = Map(Variable, Value)
op   ValuationVars (v:Valuation): Set Variable = domain(v)
```

2.4 Syntax: Constraints

Next we specify the type of Constraints and various constructors and observations of Constraints.

```
type Constraint
op varsOf: Constraint -> Set Variable
```

Details of constraints for particular problems will be specified by interpretation of `Constraint`; e.g. a problem that has first-order constraints would extend `Constraint` with additional structure:

```
type FOConstraint
op varsOf: FOConstraint -> Set Variable

op mkTrue : FOConstraint
op mkFalse: FOConstraint
op mkNegation:    FOConstraint -> FOConstraint -> FOConstraint
op mkConjunction: FOConstraint -> FOConstraint -> FOConstraint
op mkDisjunction: FOConstraint -> FOConstraint -> FOConstraint
op mkExistential: Variable -> FOConstraint -> FOConstraint
op mkUniversal:   Variable -> FOConstraint -> FOConstraint
```

2.5 Logic

We assign meaning to constraints by means of an interpreter, `satisfiesC`, that evaluates the representation for a given valuation of the variables. When convenient we use the nonASCII symbol $m \models C$ to express that valuation (or assignment) m satisfies constraint C.

```
op satisfiesC : Constraint -> Valuation -> Boolean
op satisfiable  : Constraint -> Boolean
axiom def_of_satisfiable is
  fa(p:Constraint) satisfiable(p) =
    (ex(vm:Valuation)(varsOf(p)=domain(vm) && satisfiesC p vm))
```

Given the essential logical notion of satisfaction, we can begin to build up notions of logical inference. The Resolve operator is discussed in detail in Section 3.3.

```
op Entails: Constraint -> Constraint -> Boolean
axiom def_of_Entails is
  fa(C1:Constraint,C2:Constraint)
     (Entails C1 C2) = (fa(V:Valuation)(satisfiesC C1 V => satisfiesC C2 V))

op Equivalent: Constraint -> Constraint -> Boolean
axiom def_of_Equivalent is
  fa(C1:Constraint,C2:Constraint)
     (Equivalent C1 C2) = (Entails C1 C2 && Entails C2 C1)

op Simplify: Constraint -> Valuation -> Constraint
axiom Simplify_is_partial_eval is
  fa(C:Constraint,val:Valuation)(Equivalent C (Simplify C val))

op Resolve(C1:Constraint)(C2:Constraint)
          (v:Variable|  v in? (varsOf C1) && v in? (varsOf C2) ):
          {C:Constraint |   % ‘C iff ex(v)(C1 && C2)’
                         Equivalent C (mkExistential v (mkConjunction C1 C2))
          }
```

2.6 CSP specification

To specify a CSP, we must provide an interpretation of each of the concepts above, which in Specware is expressed as a specification morphism.

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

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

op Solve (Cs: Set Constraint):
         {ov: Option Valuation |
            case ov of
              | None -> ~(satisfiableCs Cs)
              | Some vm -> satisfiesCs Cs vm}
```

Let $P = \langle Var, \{D_i\}, Cs, obj \rangle$ be a CSP. A valuation m is *feasible* if it satisfies Cs. A feasible valuation m that optimizes the objective function *obj* is called *optimal.* The set of models of a constraint set Cs is $mod(Cs)$ and the set of countermodels of $cmod(Cs)$. [1] Clearly, a set of constraints Cs is unsatisfiable exactly when $mod(Cs)$ is empty and exactly when $cmod(Cs)$ is all valuations.

2.7 Special Classes of Constraint Solving

There are several well-known classes of CSP that admit polynomial-time algorithms. We list a few prominent ones; see [1] for others.

1. *Acyclic constraint networks* – Constraints define the (hyper)arcs of a graph (aka the constraint network) and variables define the nodes Some properties of CSP can be gleaned from analyzing the topology of the constraint network. A tree-like network or a DAG can be solved directly solvable by constraint propagation, i.e. without backtracking.
2. *Linear Programming* – Variables are real-valued and the constraints are linear inequalities (or equalities). This class has a variety of efficient solvers including the Simplex algorithms and interior point methods. More generally, real convex optimization problems can be solved efficiently by interior point methods.
3. *Definite Constraint systems* – This class of CSPs is a solvable in linear time under certain assumptions [10].

The last class, of definite constraints, is especially important here since it underlies all constraint propagation algorithms known to us.

Let $\langle P, \leq \rangle$ be a poset. A *definite constraint* C over P has the form

$$\tau[V] \leq A$$

[1]Any valuation that does not satisfy the constraints is a *countermodel.*

where τ is a monotone function on P over the variables V, and A is a constant or variable. Horn clauses are an important special case: Consider the bounded semilattice $\langle Boolean, \implies, \wedge, true, false\rangle$ where definite constraints have the form $x \wedge y \wedge z \implies w$; i.e. Horn clauses. A least fixpoint algorithm provides an efficient means for solving a set of definite constraints.

The generic Definite Constraint Solver algorithm for solving a set of definite constraints [10] can be derived via fixpoint iteration and the use of Finite Differencing to incrementally compute the workset that controls the iteration.

The dual formulation of definite constraints is also useful:

$$A \leq \tau[V]$$

where the monotone function τ now provides an upper bound on atom A. A greatest fixpoint algorithm provides an efficient means for solving a set of dual definite constraints. We will use both forms in subsequent text.

Applications of definite constraint solving include many problems in program analysis (type inference usage count, dataflow, binding time, strictness), Horn-SAT, and constraint propagation algorithms in CSP.

2.8 Galois Connections

The mathematical concept of *Galois Connection (GC)* is often used to define the relationship between a domain of interest (a collection of entities) and a domain of abstract representations of them. The need for GC arises when we want (1) to focus on and reason about key features of entities, without all their detail, and (2) to work efficiently with a compact, finite representation of infinite entities. A Galois Connection provides a way to reason about a complex domain in terms of a simpler abstraction, trading precision for efficiency.

In CSP the concrete domain of interest is sets of valuations. We are especially interested in knowing if the set of models for a constraint set is empty, and if not, to find a witness/model. The powerset of the set of valuations is 2^{n^n} for n binary variables. The value of a Galois Connection is providing an abstract domain that allows sound reasoning about such a large set.

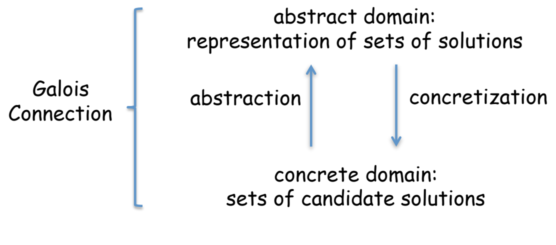

Galois Connection
abstract domain: representation of sets of solutions
abstraction
concretization
concrete domain: sets of candidate solutions

Figure 1: Galois Connection

Formally, a *Galois Connection* between "concrete" poset $\mathcal{C} = \langle C, \leq \rangle$ and "abstract" poset $\mathcal{A} = \langle A, \preceq \rangle$ is given by a pair of monotone functions $\alpha : C \to A$ (the *abstraction* function) and $\gamma : A \to C$ (the *concretization* function). Moreover, the monotone functions must satisfy the Galois Connection condition:

$$a \leq \alpha(c) \ \equiv \ \gamma(a) \preceq c.$$

Intuitively, α maps a concrete entity to the best possible abstraction in A. Conversely, γ gives the best concrete denotation of an abstract value.

We specify Galois Connections in Metaslang in a way that is specialized to the purposes of constraint solving:

```
  import Constraint#Valuation
  import translate /Library/Math/Semilattice#BoundedJoinSemilattice by
     {A      +-> Rhat,
      <=     +-> RefinesTo,
      bot    +-> RhatBot,
      join   +-> RhatJoin}

 op concretize: Rhat -> Set Valuation                % gamma
 op abstract  : Set Valuation -> Rhat                % alpha
 op beta      : Valuation     -> Rhat                % beta
 axiom Galois-Connection-Condition is
    fa(S:Set Valuation,rhat:Rhat)
       rhat RefinesTo (abstract S) = S subset (concretize rhat)
 theorem Overapprox-Galois-Connection-C is
    fa(S:Set Valuation) S subset (concretize (abstract S))
 theorem Overapprox-Galois-Connection-A is   % typically this is equality
    fa(rhat:Rhat) rhat RefinesTo (abstract (concretize rhat))
 axiom RhatBot-is-comprehensive is    % i.e. R = concretize RhatBot
   fa(z:Valuation)( z in? concretize RhatBot )
 axiom beta-isomorphism is     %  gamma beta z = {z}
   fa(z:Valuation,y:Valuation)( y in (gamma beta z) = (y=z) )
```

The extra function `beta` is useful in specifications since it maps a concrete element to an abstract element with the same meaning. In addition, the CW library includes several standard Galois Connections that are used in constraint solving, represented by morphism from this specification. These are discussed below.

A simple example of a GC is the relationship between finite sets of integers and their abstraction as bounding intervals. A set, such as {1,3,5} is abstracted to the interval $\langle 1, 5 \rangle$, given by the least and greatest elements of the set. The denotation, or concretization, of an interval $\langle 1, 5 \rangle$ is the full set of elements in the interval, {1,2,3,4,5} in this case:

$$\alpha(\{1, 3, 5\}) = \langle 1, 5 \rangle$$

$$\gamma(\langle 1, 5\rangle) = \{1, 2, 3, 4, 5\}.$$

Note that α and γ are not inverses of one another, since the abstraction loses information: here the fact that 2 and 4 are not in the set. Intuitively then, when we abstract a set and then concretize, we end up with a superset:

$$\gamma(\alpha(\{1, 3, 5\})) = \{1, 2, 3, 4, 5\} \supseteq \{1, 3, 5\}.$$

However, we would usually expect that abstracting the concretization of an interval set would return the interval:

$$\alpha(\gamma(\langle 1, 5\rangle)) = \alpha(\{1, 2, 3, 4, 5\}) = \langle 1, 5\rangle.$$

For purposes of this report, we will always take the concrete domain to be sets of valuations (e.g. a typical element being the models for a given constraint set). For example, consider a SAT problem with three binary variables x, y, z, and constraint

$$C = (x \ \vee \ \neg y \ \vee \ z) \ \wedge \ (\neg x \ \vee \ y).$$

We have

$$\begin{array}{lcl} mod(C) & = & \{000, 001, 011, 110, 111\} \\ cmod(C) & = & \{010, 100, 101\} \end{array}$$

One example of a standard abstract domain is the domain of value-set assignments, which are finite maps from Variables to sets of Values:

```
type VSA = Map(Variable, Set Value)
```

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.

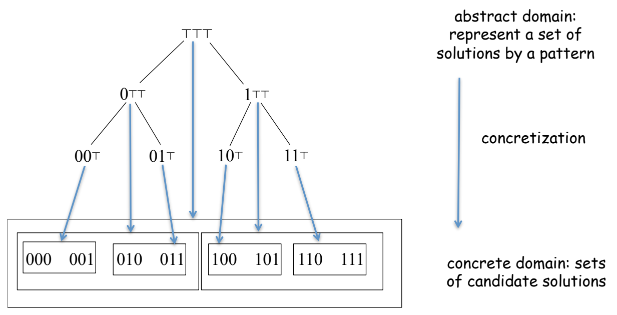
```
op RefinesTo(pv:PValuation, qv:PValuation): Boolean
axiom RefinesTo-def is
  fa(pv,qv)( RefinesTo(pv,qv) = fa(v:Variable)( pv(v) superset qv(v)) )
```

For brevity in the text, we will usually write $pv \sqsubseteq qv$ for `RefinesTo(pv,qv)`.

The abstract domain of value-set assignments is depicted in Figure 2 where unassigned values are indicated by the top symbol $\top$ (i.e. all possible values are available). For example, the partial assignment $0\top\top$ assigns 0 to variable x and has all possible values for y and z. The downward arrows are the $\sqsubseteq$ relation; e.g. $0\top\top \sqsubseteq 01\top$.

The concretization function maps each abstract domain element (i.e. partial assignment) to a set of (total) assignments; e.g.

$$\gamma(0\top\top) \ = \ \{000, 001, 010, 011\}.$$



⊤⊤⊤
0⊤⊤
1⊤⊤
00⊤
01⊤
10⊤
11⊤
000 001
010 011
100 101
110 111
abstract domain: represent a set of solutions by a pattern
concretization
concrete domain: sets of candidate solutions

Figure 2: Concretization of Partial Assignments

Note that the abstract domain loses precision because it cannot represent all subsets of assignments: there are $3^3 = 27$ abstract objects versus $2^8 = 256$ subsets of assignments. Since each of the 256 subsets of assignments will be the models of some constraint, an algorithm must be able to reason about any subset. The abstract domain can only approximate the concrete domain, the Galois Connection conditions support the notion of finding a best approximation to any subset. For example, we have

$$\alpha(\{010, 011, 111\}) \; = \; \top 1 \top$$

but notice that this is an overgeneralization in the sense that

$$\gamma(\top 1 \top) = \{010, 011, 110, 111\}$$

which is a superset of $\{010, 011, 111\}$. Generally, an *overapproximating domain* is characterized by the laws

$$asgs \subseteq \gamma \cdot \alpha(asgs)$$

and

$$\alpha \cdot \gamma(pv) \sqsubseteq pv$$

essentially because α yields a least abstraction domain value that concretizes to a superset. Dually, $\gamma(pv)$ yields a greatest set of assignments that abstracts to a lower bound on pv.

Formally we have a Galois Connection between $\langle 2^{Valuation}, \subseteq \rangle$ and
$\langle VSA = map(Variable, Set\ Value), \sqsubseteq \rangle$ given by the monotone functions

$$\alpha(asgs) = \{v \mapsto \cup\{asg(v) \mid asg \in asgs\} \mid v \in Variables\}$$

$$\gamma(pv) = \{asg \mid pv \sqsubseteq \beta asg\}.$$

Since we have

$$asgs \subseteq \gamma \cdot \alpha(asgs)$$

and

$$\alpha \cdot \gamma(pv) \sqsubseteq pv,$$

the abstract domain is *overapproximating* [2]. If the signs are reversed as in

$$\gamma \cdot \alpha(asgs) \subseteq asgs$$

and

$$pv \sqsubseteq \alpha \cdot \gamma(pv)$$

we have an *underapproximating* domain.

Further examples: the models listed above are not the concretization of any partial assignment, but

$$\begin{aligned} \alpha(mod(C)) &= \alpha(\{000, 001, 011, 110, 111\}) = \top\top\top \\ \alpha(cmod(C)) &= \alpha(\{010, 100, 101\}) = \top\top\top \end{aligned}$$

There are many other abstract domains that are possible for the concrete domain of sets of Valuations. For the SAT problem (and other 0/1 problems), it is common to use a version of the value-set domain consisting of values $\perp$, `false`, `true`, and $\top$.

2.9 Abstract Interpretation and Constraint Solving

A Galois Connection (GC) provides an abstract approximation to the concrete semantics of a CSP, but the real problem is to extract a feasible/optimal solution or to show that none exists. How does a GC help? Given a set of valuations S, we are interested in the subset of S that are models of our given constraints. We can formalize this as a concrete operation

$$mod_\theta(S) = \{m \mid m \in S \ \wedge \ m \models \theta\}$$

and its complement

$$cmod_\theta(S) = \{m \mid m \in S \ \wedge \ m \not\models \theta\}.$$

Unsatisfiability is just that $mod_\theta(S) = \{\}$, and satisfiability is producing a witness from $mod_\theta(S)$. mod_θ is a useful concrete operation, but it is typically not readily computable; in other words, it is useful for specification purposes, but not directly useful for computation. What we want is a way to approximate the effect of concrete operations. The idea of abstract interpretation is to use a Galois Connection to specify an abstract operation that reflects the effect of a concrete operation, particularly *mod* and *cmod*.

Figure 3 shows two characterizations of abstract operations that approximate concrete operations. The general case is shown on the right and defines the notion of a sound overapproximation f_A to the concrete operation f_C:

$$f_C \cdot \gamma(a) \leq \gamma \cdot f_A(a).$$

[2] Here we actually have the stronger property $\alpha \cdot \gamma(pv) = pv$, in which case the Galois Connection is known as a *Galois Insertion*.

The figure on the left shows the ideal special case. A *best* abstract operator returns the maximal amount of information possible within the abstract domain. These characterizations are useful for specification purposes. Even though the characterization of a best abstract operator provides an equation, it is not finitely computable since the intermediate sets are typically extremely large or infinite. The purpose of the abstraction is to sacrifice some precision to gain performance.

$$\begin{tikzcd}
a \arrow[r, "f_A"] \arrow[d, "\gamma"'] & a' \\
c \arrow[r, "f_C"] & c' \arrow[u, "\alpha"']
\end{tikzcd}
\qquad
\begin{tikzcd}
a \arrow[r, "f_A"] \arrow[d, "\gamma"'] \arrow[dr, phantom, "\leqslant"] & a' \arrow[d, "\gamma"] \\
c \arrow[r, "f_C"] & c'
\end{tikzcd}$$

best abstract operator:
$f_A(a) = \alpha \cdot f_C \cdot \gamma(a)$

sound abstract operator:
$f_C \cdot \gamma(a) \leqslant \gamma \cdot f_A(a)$

Figure 3: Abstract Operations

2.10 Abstract Operators: Constraint Propagation

We can now tie together two previously introduced topics: definite constraint systems and abstract operators. *The result is the first general method for generating sound propagation operators for CSPs over arbitrary logics.*

In CSP theory [1] the most basic sound abstract operator is known as Arc-Consistency (AC) (where "arc" refers to the arcs of a binary constraint network). Let $C(x, y)$ be a binary constraint over variables x and y, and let vs denote the current value-set assignment, so $vs(x)$ is the set of possible values for var x, i.e. the current value set for x.

A binary constraint $C(x, y)$ is *arc-consistent* if

1. for every x value in $vs(x)$, there is a y value in $vs(y)$ that satisfies C
2. for every y value in $vs(y)$, there is a x value in $vs(x)$ that satisfies C

These two conditions can be expressed concisely using relational calculus operations:

$$\pi_x \cdot (vs(y) \bowtie C) \supseteq vs(x)$$

$$\pi_y \cdot (vs(x) \bowtie C) \supseteq vs(y)$$

where we treat the binary constraint C as a binary relation, $\bowtie$ is the relational join operator, and π_z projects a relation along variable z. Note that these are definite constraints over the domains $vs(v)$: the lhs is a monotone function of the current value sets and provides a bound on a variable's value set.

We can automatically generate these AC constraints from the given constraint set, yielding a set of definite constraints that can be solved using the Definite Constraint Solver algorithm discussed in Section 2.7. Note that the iteration will compute a greatest fixpoint of the variable value sets, iterating downward from the top of the value set lattice.

Arc consistency can be readily generalized to n-ary constraints; e.g. for a ternary constraint $C(x, y, z)$,

$$\pi_x \cdot (vs(y) \times vs(z) \bowtie C) \supseteq vs(x).$$

We prove next that AC provides a best abstract operator for the concrete operation of filtering out models of a given constraint C.

Theorem 2.1 *Arc Consistency defines a best abstract transformer for mod_C for any constraint C in any logic.*

Proof: Without loss of generality, consider a binary constraint $C(x, y)$. The proof is similar for arbitrary n-ary constraints. Let $vs(x)$ and $vs(y)$ be the current value sets for variables x and y respectively. Arc consistency defines two definite constraints

$$\pi_x \cdot (vs(y) \bowtie C) \supseteq vs(x)$$

$$\pi_y \cdot (vs(x) \bowtie C) \supseteq vs(y)$$

and the Definite Constraint Solver algorithm effectively iterates the following monotone function:

```
$AC(vs, C)$ = if $eval(vs, C) = \texttt{False}$
              then $\top$
              else $vs(x) \leftarrow vs(x) \bigcap \pi_x(vs(y) \bowtie C)$;
                   $vs(y) \leftarrow vs(y) \bigcap \pi_y(vs(x) \bowtie C)$.
```

where $eval(vs, C)$ is a necessary test for the satifiability of C given the partial valuation vs:

$$\exists z(z \in \gamma(vs) \ \wedge \ C(z)) \implies eval(vs, C)$$

so that $eval(vs, C) = \texttt{False}$ implies that C is unsatisfiable under any refinement to vs. Typically this test is implemented by simplifying C after performing the instantiations defined in vs.

Our goal is to prove that $AC(vs, C) = \alpha \cdot mod_C \cdot \gamma(vs)$; i.e. that AC computes a best abstract operator for mod_C. We proceed by cases. First, when $eval(vs, C) = \texttt{False}$, then $mod_C \cdot \gamma(vs)$ is empty, and then $\alpha(\{\})$ is $\top$. At the same time, $AC(vs, C)$ evaluates to $\top$. Second, when $eval(vs, C) \neq \texttt{False}$:

$$\begin{aligned}
&(\alpha \cdot mod_C \cdot \gamma(vs))(x) && \\
&\quad = \ \alpha(\{m \mid m \in \gamma(vs) \ \wedge \ m \models C\})(x) && \text{by definition of } mod_C(X) \\
&\quad = \ \cup \{m(x) \mid m \in \gamma(vs) \ \wedge \ m \models C\} && \text{using the definition of } \alpha \text{ applied to } x \\
&\quad = \ \cup \{m(x) \mid m \in \gamma(vs)\} \ \cap \ \cup \{m(x) \mid m \in \gamma(vs) \ \wedge \ m \models C\} && \text{rearranging terms} \\
&\quad = \ vs(x) \ \cap \ \pi_x(vs(y) \bowtie C) && \text{simplifying} \\
&\quad = \ AC(vs, C)(x). && \text{by definition}
\end{aligned}$$

A similar calculation applies to y. $\square$

We show below that the Unit Rule in SAT, temporal propagation in scheduling, linear resolution in ILP are instances of AC. In fact, all sound propagation operators known to us are instances of AC. The theorem asserts the optimality of these operators in that they reflect the maximum possible information from the concrete constraints to the abstract domain representation.

From Arc Consistency to Definite Constraints Since the AC condition characterizes best abstract operators in CSP, we want to use those conditions as a generic (logic-independent) starting point for calculating propagation rules. The relational calculus form of the AC is too complex for computation, since it involves large or infinite sets. However, we can often transform the logical formulation of AC into simply implementable definite constraints as follows. The logical formulation of arc-consistency for a binary constraint C over variables x and y is:

$$\forall xv(xv \in vs(x) \implies \exists yv(yv \in vs(y) \ \wedge \ C(xv, yv)))$$

$$\forall yv(yv \in vs(y) \implies \exists xv(xv \in vs(x) \ \wedge \ C(xv, yv))).$$

For a ternary constraint $C(x, y, x)$, we can generate the form

$$\forall xv(xv \in vs(x) \implies \exists yv \exists zv(yv \in vs(y) \ \wedge \ zv \in vs(z) \ \wedge \ C(xv, yv, zv))).$$

and so on.

To calculate a useful form for these definite constraints, we need (1) a representation of the variable domain types $vs(v)$, and (2) a representation of the constraint C. We apply quantifier elimination laws from the underlying constraint logic to derive definite constraints on the abstract value-set domain. See Appendix A for a list of the quantifier elimination laws used.

Examples

Each n-ary concrete constraint gives rise to n definite constraints on the abstract domain. Collecting these definite constraints from all concrete constraints defines a CSP for propagation that can be solved by fixpoint iteration. The result is a maximally refined abstract value that reflects the concrete semantics of modC. We apply the above method to the following examples: scheduling, SAT, and Linear pseudo-Boolean.

Example: Temporal Propagation in Scheduling

The start time of a task (concrete domain) is often represented by a time window in the abstract domain: $\langle earliestStartTime, latestStartTime \rangle$, abbreviated $\langle est, lst \rangle$. That is, if st_i is the (concrete, unknown) start time of task i, then let $est_i \leq st_i \leq lst_i$ be the (abstraction domain) bounds on st_i.

A typical constraint expresses that task i must finish before task $i + 1$ can start:

$$st_i + duration_i \leq st_{i+1}.$$

This binary constraint gives rise to two AC forms. We instantiate and simplify one of them:

$$\forall st_1(est_1 \le st_1 \le lst_1 \implies \exists st_2(est_2 \le st_2 \le lst_2 \ \wedge \ st_1 + duration \le st_2))$$

$$= \qquad \{ \text{ Quantifier Elimination law 2.1 on monotone } st_2 \ \}$$

$$\forall st_1(est_1 \le st_1 \le lst_1 \implies (est_2 \le lst_2 \ \wedge \ st_1 + duration \le lst_2))$$

$$= \qquad \{ \text{ Quantifier Elimination law -2.1 on antitone } st_1\text{, simplify } \}$$

$$est_1 \le lst_1 \implies lst_1 + duration \le lst_2$$

$$= \qquad \{ \text{ simplify } \}$$

$$lst_1 + duration \le lst_2.$$

Focusing on the other AC form produces a corresponding definite constraint: $est_1 + duration \le est_2$. The calculation yields simple definite constraints that can be used to propagate the effect of decisions on time windows, ensuring that the current partial valuation is temporally consistent.

Example: Unit Rule in SAT

A typical SAT constraint is $x \vee y \vee \neg z$. This constraint has three variables, so we instantiate the ternary AC forms; e.g.

$\forall x(x \in vs(x) \implies \exists y \exists z(y \in vs(y) \ \wedge \ z \in vs(z) \ \wedge \ C(x, y, z)))$

where $vs(x)$ is the current value set for variable x, Here we chose to represent the value set for a variable x by a lower/upper bound pair $\langle x_l, x_u \rangle$, which is initially $\langle \texttt{False}, \texttt{True} \rangle$, with implication as the partial order. The instantiated AC constraint is:

$$\forall x(x_l \le x \le x_u \implies \exists y \exists z(y_l \le y \le y_u \ \wedge \ z_l \le z \le z_u \ \wedge \ (x \ \vee \ y \ \vee \ \neg z)))$$

$$= \qquad \{ \text{ QE law 2.1 applied to } y \ \}$$

$$\forall x(x_l \le x \le x_u \implies \exists z(y_l \le y_u \ \wedge \ z_l \le z \le z_u \ \wedge \ (x \ \vee \ y_u \ \vee \ \neg z)))$$

$$= \qquad \{ \text{ QE law -2.1 applied to } z\text{, simplify } \}$$

$$\forall x(x_l \le x \le x_u \implies (z_l \le z_u \ \wedge \ (x \ \vee \ y_u \ \vee \ \neg z_l)))$$

$$= \qquad \{ \text{ QE law 2.2 applied to } x\text{, simplify } \}$$

$$x_l \ \vee \ y_u \ \vee \ \neg z_l$$

$$= \qquad \{ \text{ convert to definite form: } x_l \text{ has positive polarity, } \neg z_l \text{ and } y_u \text{ have negative polarity } \}$$

$\neg y_u \wedge z_l \implies x_l.$

The instances of the other AC forms produce the other Unit Rules for this clause.

Example: Propagation in 0,1-ILP

A constraint in 0,1-Integer Linear Programming (0,1-ILP) has the form

$$a_1 \cdot x_1 + a_2 \cdot x_2 + \ + a_n \cdot x_n \geq b$$

where for $i = 1, 2, n$, x_i is a literal, and b, a_i are positive constants.

If consider the example

$$5 \cdot x_2 + 4 \cdot \neg x_3 + 2 \cdot x_4 + x_5 \geq 9$$

and bounds $x_{il} \leq x_i \leq x_{iu}$ for $i = 2, 3, 4, 5$ and focus on variable x_2, then using the same calculation style the Arc Consistency instance simplifies to

$$9(4 \cdot \neg x_{3l} + 2 \cdot x_{4u} + x_{5u})/5 \leq x_{2l}$$

which forces x_{2l} up when x_{3l} increases, x_{4u} decreases, and x_{5u} decreases.

Analogous definite constraints result from instantiating the other 3 AC forms.

A fully realized synthesis system would automatically (1) lift concrete constraints to abstract definite constraints and (2) construct a custom instance of the Definite Constraint Solver algorithm, resulting in an optimal constraint propagation algorithm. Note again that the entire derivation is specified in a logic-dependent way.

3 Algorithm Development

3.1 Algorithm Theory

We present the current version of our algorithm theory for Global Search with Conflict-Directed Backjumping and Learning applied to Constraint Satisfaction Problems. Any algorithm theory is expressed as a parametric specification where the parameter specification is interpreted via a classification morphism into the application domain specification. We instantiate the algorithm theory by taking a pushout of the classification morphism and the parameter injection.

`GS_CDBL` theory is parametric on an overapproximating abstract domain for search and propagation. The concrete domain is fixed as (Set Valuation). The algorithm theory specification is intended to provide enough detail to support an abstract proof of correctness of the algorithm theory.

```
GS_CDBL_Theory = spec
  import ProblemTheoryC#CSP,       % constraint satisfaction problem thy
         GS_Galois_Connection,     % Rhat: overapproximating abstract domain
         /Library/DataStructures/StructuredTypes

  type State      % problem-solving state
  type InferenceStack
  type RefinementReason   % sum type

  op input             : State -> D
  op constraints       : State -> Set Constraint
  op currentDepth      : State -> Nat
  op currentRhat       : State -> Rhat
  op currentInferences : State -> InferenceStack
  op bindingDepth      : State -> Map(Variable, Nat)
  op stk               : State -> Map(Nat, Rhat*InferenceStack)

  op initialVariables   : D -> Set Variable

  op phi : D*Rhat -> Bool
  axiom characterization_of_necessary_pruning_test_phi is
     fa(x:D,z:R,st:State)(z in? (concretize (currentRhat st)) && (O x z)
                           => phi (input st,currentRhat st))
  op psi : Rhat -> Rhat            % necessary propagator
  op [a] monotone? (f:a->a, le:a*a->Bool):Bool  % needs axiom, not def

  axiom characterization_of_necessary_propagator_psi is
    monotone?(psi, RefinesTo) &&
    (fa(x:D,r:Rhat,z:R)(z in? (concretize r) && (O x z)
                       => (z in? concretize (psi r))))

  op xi : Rhat -> Rhat         % consistent refinement
  axiom characterization_of_consistent_refinement_xi is
    monotone?(xi, RefinesTo) &&
    (fa(x:D,r:Rhat) (ex(z:R)(z in? (concretize r)      && (O x z))
                = (ex(z:R)(z in? (concretize (xi r)) && (O x z)))))
  end-spec
```

State is introduced to package up the local state of the solver, including the stack of subspaces and the record of the chain of inferences performed by propagation.

phi is a pruning test. It is a necessary condition on existence of a feasible solution in the current subspace. For CSPs in which constraints are given as input data, this test is simply that no constraint fails under the current partial assignment.

psi is a monotone function on spaces that adds necessary constraints to the current space. By construction it preserves all feasible solutions. This spec for psi is equivalent to the Abstract Interpretation characterization of a sound abstract operator (for the concrete operator that filters for models). Monotonicity allows us to iterate it to a fixpoint. We can generate psi by a standard procedure to formulate and simplify arc-consistency variants of each constraint, yielding a definite constraint system.

xi is a monotone function that generates a consistent refinement of a space. By construction it preserves the existence of feasible solutions.

```
GS_CDBL = spec
  import GS_CDBL_Theory

  op InitialState(x:D):{st:State |
                          input st = x
                          && constraints st = initialConstraints x
                          && currentDepth st = 0
                          && currentInferences st = empty_stack
                          && stk st = empty_map}

  type PropagateFailInfo = Constraint
  type PropagateResult  = | Ok State | Fail State*PropagateFailInfo

  op Propagate (st:State | phi (input st,currentRhat st)):
              {pr:PropagateResult |
                  (case pr of
                     | Ok st' -> RefinesTo(currentRhat st, currentRhat st')
                                 && input st' = input st
                                 && currentDepth st' = currentDepth st
                                 && psi(currentRhat st') = currentRhat st'
                                 &&  xi(currentRhat st') = currentRhat st'
                                 && phi(input st, currentRhat st')
                     | Fail (st',cc) ->
                                 RefinesTo(currentRhat st, currentRhat st')
                                 && input st' = input st
                                 && currentDepth st' = currentDepth st
                                 && ~(phi(input st',currentRhat st'))
                                 && cc in? constraints st'
                                 && ~(satisfiesRhat (currentRhat st') cc)
                                 )
              }

  op AnalyzeForCompleteness(st:State | phi (input st,currentRhat st)):
       {er:AFCResult |
             let rhat = (currentRhat st) in
```

```
           case er of
             | Answer z -> (O (input st) z)
             | Extrapolate (vr,vl) ->
                 RefinesTo(rhat, RhatJoin(rhat,beta (singletonMap vr vl)))
                 && phi((input st),RhatJoin(rhat,beta (singletonMap vr vl)))
             | Fail lc ->
                 (fa(shat)(RefinesTo(rhat,shat) => ~(phi(input st,shat))))
                 && ~(satisfiesRhat rhat lc)
                 && (entailsCs (constraints st) lc)}

type AFCResult = | Answer R | Fail Constraint | Extrapolate ExtrapolateInfo
type PEResult  = | Answer R | Fail (State*PropagateFailInfo)
type ExtrapolateInfo = Variable*Value
op incorporateE(rhat:Rhat)((vr,vl):ExtrapolateInfo): Rhat
        = RhatJoin(rhat, beta(singletonMap vr vl))

type Inference = {assignee:Variable, depth:Nat, reason:RefinementReason}
type InferenceStack = Stack Inference

op EnforceE(st :State)((vr,vl):ExtrapolateInfo)
         :{st':State | currentRhat st' = incorporateE(currentRhat st)(vr,vl)
               && currentDepth st' = (currentDepth st) + 1
               && currentInferences st' = empty_stack
               && bindingDepth st' = update (bindingDepth st) vr
                                         (1 + currentDepth st)
               && stk st' = (update (stk st) (currentDepth st)
                                   (currentRhat st, currentInferences st))}
```

```
op satisfiesRhat(rhat:Rhat)(c:Constraint):Bool % abstractSatisfies?
axiom satisfiesRhat-def is
    fa(rhat:Rhat,c:Constraint)
    ex(z:Valuation)(z in? (concretize rhat) && (satisfiesC c z))

type ConflictConstraint = Constraint
op refinable(st:State)(pc:ConflictConstraint)(bjd:Nat):Bool

op AnalyzeConflict(st:State)
                 (cc:PropagateFailInfo | cc in? constraints st
                                   && ~(satisfiesRhat(currentRhat st) cc)):
    {opcc:Option (ConflictConstraint*Nat)
      | case opcc of
         | Some (lc,bjd) -> (entailsCs (constraints st) lc)
                          && ~(satisfiesRhat(currentRhat st) lc)
```

```
                           && (refinable st lc bjd)
         | None         -> ~(ex(lc:ConflictConstraint,d:Nat)
                              (refinable st lc d))}

op EnforceC(st :State)(bjd:Nat)
           (lc:ConflictConstraint | ~(lc in? (constraints st))):
           {st':State | currentDepth st' = bjd
                     && constraints st' = set_insert_new(lc, constraints st)
                     && currentRhat st' = (TMApply(stk st, bjd)).1
                     && currentInferences st' = (TMApply(stk st, bjd)).2
                     && bindingDepth st' = bindingDepth st
            }

op PropagateExtrapolate(st:State | phi (input st,currentRhat st)):
  {per:PEResult |
`            case per of
              | Answer z       -> (O (input st) z)
              | Fail (st',cc)  -> RefinesTo(currentRhat st, currentRhat st')
                                  && input st' = input st
                                  && currentDepth st' >= currentDepth st
                                  && ~(phi (input st', currentRhat st'))
                                  && (entailsCs (constraints st) cc)
                                  && ~(satisfiesRhat (currentRhat st') cc)}
   = case Propagate st of
      | Ok st1 -> (case AnalyzeForCompleteness st1 of % extract a solution?
                     | Answer z -> Answer z           % if so, we're done
                     | Extrapolate (vr,vl) ->
                           let st2 = EnforceE st1 (vr,vl) in
                           PropagateExtrapolate st2
                     | Fail cc -> Fail (st1,cc))
      | Fail stcc -> Fail stcc

def GS (st:State | phi (input st,currentRhat st) ):
       {ov:Option R | feasible (input st) ov } =
  case PropagateExtrapolate st of                   % propagate; complete?
    | Answer z ->  Some z                           % if success, we're done
    | Fail (st1, cc) ->                             % constraint cc fails
       (case AnalyzeConflict st1 cc of              % analyze failure
        | Some (lc,bjd) -> GS (EnforceC st1 bjd lc) % learn lc, backjump
        | None         -> None)                     % no solutions

op feasible(x:D)(ov: Option R): Bool =
     case ov of
       | None -> ~(ex(z)(O x z))
       | Some z -> (O x z)
  def Solve (x:D): {ov: Option R | feasible x ov } =
      let is = InitialState x in
      if phi(x, currentRhat is) then (GS is) else None

end-spec
```

Propagate computes a refinement of the current space by iterating psi and xi to a (least) fixpoint. The properties of psi and xi are preserved under iteration. At each iteration, propagate applies phi to test for emptiness. Key enablers are that psi and xi are monotone functions of rhat, and phi is antitone.

Propagate iterates abstract operators psi and xi until either (Ok case) a fixpoint it reached, or (Fail case) we have that the current partial assignment (rhat) does not satisfy some constraint.

AnalyzeForCompleteness attempts to extract a feasible solution from rhat and return it as the answer. Otherwise, analyze the reason for the failure of extract (typically as a disjunction) and use it to pick a disjunct from the alternatives and refine currentRhat by enforcing the disjunct. If there is no feasible refinement of rhat then prune it (i.e. fail).

AnalyzeConflict analyzes the reason for the failure of the current partial solution; return a new constraint and backjump depth if there is a depth at which a proper propagation refinement can occur (satisfying phi), else return None signifying unsatisfiability (i.e. false is a consequence of the current constraints). The learned constraint satisfies: (1) it is a consequence of the current constraints, (2) it fails the conflict constraint, and (3) it allows propagation inference at an earlier depth. The backjump depth bjd should be the least depth at which a propagation step could be made (although more generally, it should be the least depth at which the search would take a different direction than if the lc were not learned; e.g. if a heuristic would make a different choice. This is effectively restarting.

3.2 Correctness of the Algorithm Theory

We proceed top-down. We first show that the definition of Solve satisfies the theorem if GS satisfies its specification. Then we show that the definition of GS satisfies its spec if its subfunctions satisfy their specs, and so on until we ground out in subfunctions with specs but no defs. We have proofs for Solve, GS, and PropagateExtrapolate. The remaining functions are given problem-specific specifications (i.e. pre/post-conditions) after composition of CDBL with the problem domain theory, and the derivation must then synthesize definitions to satisfy those specifications.

```
Theorem 3.1.  The definition of Solve satisfies its specification, if GS
satisfies its spec.
```

```
 op feasible(x:D)(ov: Option R): Bool =
       case ov of
         | None -> ~(ex(z)(O x z))
         | Some z -> (O x z)

  def  Solve (x:D): {ov: Option R | feasible x ov } =
      let is = InitialState x in
      if (phi (currentRhat is)) then (GS is) else None

Proof: Let x:D.  We show that the definition of Solve satisfies its
postcondition.

 feasible x (Solve x)

  = { unfold Solve, and let 'is' denote the value InitialState x }

 feasible x (if (phi (currentRhat is)) then (GS is) else None)

Case 0: assume (phi (currentRhat is))

       = { evaluate using assumption }

    feasible x (GS is)

       = { assume GS satisfies its postcondition: feasible x (GS is) }

    true.

Case 1: assume ~(phi (currentRhat is)).  Then by def of phi we have
       ex(z:R)(z in? (concretize (currentRhat is)) && (O x z))
       => phi (currentRhat is),
       or by the contrapositive:
       ~ex(z:R)(z in? (concretize (currentRhat is)) && (O x z))
         <= ~phi (currentRhat is)
      so we can infer the additional assumption that
      ~ex(z:R)(z in? (concretize (currentRhat is)) && (O x z)).
      which is equal to (note that (currentRhat is) = r0hat)
      ~ex(z:R)(z in? (concretize r0hat) && (O x z)).
      which by by axiom r0hat_is_comprehensive simplifies to
      ~ex(z:R)( O x z).

   feasible x (if (phi (currentRhat is)) then (GS is) else None)

        = { evaluate using assumption }
   feasible x None

       = { unfold feasible and evaluate }

   ~(ex(y) (O x y))

       = { since the goal matches the inferred assumption }

    true.

QED
```

```
Theorem 3.2: The definition of GS satisfies its specification if
PropagateExtrapolate, AnalyzeConflict, and EnforceC satisfy their specs.

GS (st:State | phi (input st,currentRhat st) ):
   {ov:Option R | feasible (input st) ov } =
    case PropagateExtrapolate st of
      | Answer z ->  Some z
      | Fail (st1, cc) ->
         (case AnalyzeConflict st1 cc of
          | Some (p,bjd) -> let st2 = EnforceC st1 bjd p in
                             GS st2
          | None         -> None)

Proof: Let st be a State value that satisfies the precondition (phi (input
st,currentRhat st)) and let its parts be {input=x, currentRhat=rhat,...}.
Since (phi (currentRhat st)) holds, we can legally call PropagateExtrapolate,
and since it satisfies its spec by assumption, we have two cases.  In each
case we must show that the postcondition of GS holds.

 Case 1: PropagateExtrapolate st = Answer z;
         then we have (O x z) by assumption that PropagateExtrapolate
         satisfies its spec.

  feasible x (GS st)
    = {unfold & simplify using case assumption}
  feasible x (Some z)
    = {unfold feasible}
  O x z
    = {by derived consequence of the case assumption}
  true.
```

```
Case 2: PropagateExtrapolate st = Fail (st1,cc);
        then we have (by assumption that PE satisfies its spec):
                RefinesTo(currentRhat st, currentRhat st1)
                && input st1 = input st
                && currentDepth st1 = currentDepth st
                && ~(phi (currentRhat st1))
                && cc in constraints st1
                && ~(satisfiesRhat (currentRhat st') cc)

 feasible x (GS st)

   = {unfold & simplify using case assumption}

 feasible x (case AnalyzeConflict st1 fi of
            | Some (p,bjd) -> let st2 = EnforceC st1 bjd p in GS st2
            | None         -> None)

Case 2.1  AnalyzeConflict st1 cc = Some (lc,bjd);
       then we have
           entailsCs (constraints st) lc
           && ~(SatisfiesRhat(currentRhat st) lc)
           && (refinable st lc bjd)
       by our assumption that AC satisfies its spec;

     = { evaluate using case assumption }

   feasible x (GS (EnforceC st1 bjd lc))

     = { evaluate EnforceC: return a new state st2 with new depth and
       extended constraint set. }

   feasible x (GS st2)

     = { By construction we know that the current rhat will satisfy phi,
       since we only push feasible rhat on the stack.  So we can call GS and
       we can apply the induction hypothesis, so feasible is satisfied. What
       is the wfo of the induction?  }

   true.

Case 2.2  AnalyzeConflict st1 cc = None;
       then we have
           ~(ex(lc:ConflictConstraint,d:Nat)(refinable st lc d))

       by our assumption that AC satisfies its spec; in particular, the root
```

```
      node at depth 0 is not refinable.  Since the root node concretizes to
      all elements of R and inference preserves all (or existence of)
      feasible solutions, we know that there can be no feasible, so
      ~ex(z)(O x z)

  feasible x (case AnalyzeConflict st1 cc of
               | Some (p,bjd) -> let st2 = EnforceC st1 bjd p in
                                 GS st2
               | None         -> None)

     = { evaluate AnalyzeConflict }

  feasible x None
     = { unfold }
  ~(ex(z)(O x z))
     = { matching with derived assumption }
  true.
```

```
Theorem 3.3: The definition of PropagateExtrapolate satisfies its
specification if Propagate, AnalyzeForCompleteness and EnforceE satisfy their
specs.

  op PropagateExtrapolate(st:State | phi (input st,currentRhat st)):
     {per:PEResult | case per of
                     | Answer z      -> (O (input st) z)
                     | Fail (st',cc) ->
                         RefinesTo(currentRhat st, currentRhat st')
                         && input st' = input st
                         && currentDepth st' > currentDepth st
                         && ~(phi (currentRhat st'))
                         && cc in? constraints st
                         && (fa(z:R)(z in? concretize (currentRhat st)
                                   => ~SatisfiesC(z, cc)))}
     = case Propagate st of
        | Ok st1 -> (case AnalyzeForCompleteness st1 of
                     | Answer z -> Answer z
                     | Extrapolate (vr,vl) ->
                          let st2 = EnforceE st1 (beta(singletonMap vr vl)) in
                          PropagateExtrapolate st2
                     | Fail stpf -> Fail stpf)
        | Fail stcc -> Fail stcc

  type PEResult = | Answer R | Fail (State*Constraint)
```

```
Proof: Let st be a State value that satisfies the precondition
(phi (input st,currentRhat st)) and let its parts be
{input=x, currentRhat=rhat,...}.  Since (phi (currentRhat st)) holds, we can
call Propagate, and since it satisfies its spec by assumption, we have two
cases.  In each case we must show that the postcondition of
PropagateExtrapolate holds.  To simplify the presentation, we name the
postcondition of PropagateExtrapolate as feasiblePE:

  feasiblePE st per = case per of
                       | Answer z      -> (O (input st) z)
                       | Fail (st',cc) ->
                           RefinesTo(currentRhat st, currentRhat st')
                           && input st' = input st
                           && currentDepth st' >= currentDepth st
                           && ~(phi (currentRhat st'))
                           && cc in? constraints st
                           && ~(SatisfiesRhat (currentRhat st') cc)

  feasiblePE st (PropagateExtrapolate st)

    = { evaluate }

  feasiblePE st (case Propagate st of ...)

 Case 1: Propagate st = Ok st1; then since we assume that Propagate satisfies
         its spec, we know that psi (and xi) have reached a feasible fixpoint,
         and phi holds:

                       RefinesTo(currentRhat st, currentRhat st')
                       && input st' = input st
                       && currentDepth st' = currentDepth st
                       && psi(currentRhat st') = currentRhat st'
                       &&  xi(currentRhat st') = currentRhat st'
                       && phi(input st',currentRhat st')

 Case 1.1 AnalyzeForCompleteness st1 = Answer z
        then by assumption that AnalyzeForCompleteness for satisfies its spec,
        we have (O (input st) z).

  feasiblePE st (case Propagate st of ...)

    = { evaluate using assumption }

  feasiblePE st (case AnalyzeForCompleteness st1 of ...)
  = { evaluate using assumption }

feasiblePE st (Answer z))

  = { unfold and simplify feasiblePE }

(O (input st) z)

  = { matching the derived assumption }

true.


Case 1.2  AnalyzeForCompleteness st1 = Extrapolate ehat;
       then we have
            RefinesTo(rhat, RhatJoin(rhat,ehat))
            && phi(input st,RhatJoin(rhat,ehat))
       and we return per' = PropagateExtrapolate (EnforceE st1 ehat)
       which by assumption satisfies the postcondition of PE.

 feasiblePE x (case AnalyzeForCompleteness st1 of
              | Answer z -> Answer z
              | Extrapolate ehat -> let st2 = EnforceE st1 ehat in
                                          PropagateExtrapolate st2
              | Fail stpf -> Fail stpf)

   = {evaluate: AFC returns (Extrapolate ehat)}

 feasiblePE x (PropagateExtrapolate (EnforceE st1 ehat))

   = { unfold EnforceE }

 feasiblePE x (PropagateExtrapolate st2)
   where st2 = {input=x,
                currentRhat = (rhat RhatJoin ehat),
                currentDepth=(currentDepth st1)+1,
                stk = update (stk st1) (currentDepth st1) (currentRhat st1)}

   = { We can call PropagateExtrapolate since, by derived assumption we have
       phi (rhat RhatJoin ehat), so assuming by induction hypothesis that
       PropagateExtrapolate satisfies its spec, we have
           feasiblePE x (PropagateExtrapolate st2). }

  true.
```

```
Case 1.3  AnalyzeForCompleteness st1 = Fail cc;
       then we have
            fa(shat)(RefinesTo(rhat,shat) => ~(phi shat))
            && ~(satisfiesRhat rhat cc)
            && (entailsCs (constraints st) lc)
       which implies by construction of phi and distributivity of concretize
       that
            ~ex(z)( z in? concretize rhat && (O x z))
       and in particular
            fa(z)( z in? concretize rhat => ~ satisfiesC z cc)

 feasiblePE x (case AnalyzeForCompleteness st1 of
             | Answer z -> Answer z
             | Extrapolate ehat -> let st2 = EnforceE st1 ehat in
                                       PropagateExtrapolate st2
             | Fail cc -> Fail (st1,cc))

   = {evaluate: AFC returns (Fail (st1,cc))}

 feasiblePE x (Fail (st1,cc))

   = { unfold feasiblePE }

 RefinesTo(currentRhat st, currentRhat st1)
 && input st1 = input st
 && currentDepth st1 >= currentDepth st
 && ~(ex(z)(z in? concretize (currentRhat st') && (O (input st') z)))
 && (entailsCs (constraints st) lc)
 && ~(satisfiesRhat (currentRhat st') cc)

  = { we need to discharge each of these conditions.
      First: from the postcondition of Propagate we have
      RefinesTo(currentRhat st, currentRhat st1)       }

 true
 && input st1 = input st
 && currentDepth st1 >= currentDepth st
 && ~(ex(z)(z in? concretize (currentRhat st') && (O (input st') z)))
 && (entailsCs (constraints st) lc)
 && ~(satisfiesRhat (currentRhat st') cc)

  = { Second: from the postcondition of Propagate we have
      input st1 = input st                               }

 true
 && true
 && currentDepth st1 >= currentDepth st
 && ~(ex(z)(z in? concretize (currentRhat st') && (O (input st') z)))
 && (entailsCs (constraints st) lc)
 && ~(satisfiesRhat (currentRhat st') cc)

  = { Third: from the postcondition of Propagate we have
      currentDepth st1 = currentDepth st
      which implies the postcondition  }

 true
 && true
 && true
 && ~(ex(z)(z in? concretize (currentRhat st') && (O (input st') z)))
 && (entailsCs (constraints st) lc)
 && ~(satisfiesRhat (currentRhat st') cc)

  = { Fourth: the derived assumption above matches the condition }

 true
 && true
 && true
 && true
 && (entailsCs (constraints st) lc)
 && ~(satisfiesRhat (currentRhat st') cc)

  = { Fifth: the postcondition of AFC matches the fifth condition }

 true
 && true
 && true
 && true
 && true
 && ~(satisfiesRhat (currentRhat st') cc)

  = { Sixth: the postcondition of AFC matches the sixth condition }

 true.

Case 2: Propagate st = Fail (st',cc)
        then since Propagate satisfies its spec, we know the following:
           RefinesTo(currentRhat st, currentRhat st')
           && input st' = input st
           && currentDepth st' = currentDepth st
           && ~(phi(input st',currentRhat st'))
```

```
          && cc in? constraints st'
          && ~(satisfiesRhat (currentRhat st') cc)

       and we must prove the postcondition of PE holds:

  feasiblePE st (PropagateExtrapolate st)

  = { evaluate }

feasiblePE st (case Propagate st of ...)

  = { evaluate using case assumption Propagate st = Fail (st',cc) }

RefinesTo(currentRhat st, currentRhat st')
 && input st' = input st
 && currentDepth st' >= currentDepth st
 && ~(phi (currentRhat st'))
 && cc in? constraints st
 && ~(SatisfiesRhat (currentRhat st') cc)

 = {  matching with the postconditions of Propagate }

true.
```

3.3 Constraint Resolution and Conflict Analysis

The key performance breakthroughs in SAT solvers over the last 20 years have come though the `AnalyzeConflict` operation. Much of our effort in this project was focused on developing an abstract understanding of conflict analysis that is independent of the constraint logic and could be formalized in the algorithm theory. There is a worldwide flurry of theorizing on exactly this point, trying to find an abstraction of the SAT experience that can transfer readily to other problems.

The conflict analysis problem is this. Given a failure in the search for a feasible solution, find a reason for the failure and use that reason to avoid that failure and similar failures in the future. In a little more detail, find a constraint which, if it had been present in the given constraints, would have precluded the failed search step. Moreover, it is desirable to find the most general reason, or dually the strongest constraint, in order to focus the search process as mush as possible. We present below two related but different precise formulations of conflict analysis.

At the point of a search failure, we have a value-set assignment vs and a constraint $C_n \in Cs$ that fails: $vs \models C_n = \texttt{False}$.

Conflict Analysis as an under-approximating Galois Connection

It is tempting to cast conflict analysis in dual terms to the propagation inference theory developed earlier. That is, to model conflict analysis in terms of a Galois Connection and iteration within an abstract interpretation framework to capture the search for a most general reason for the failure [2, 3].

The reasons for the failure point can be formulated as follows:

$$\mathit{Reasons}(vs, C_n) \;=\; \{ws \mid \gamma(vs) \subseteq \gamma(ws) \subseteq \mathit{cmod}_{Cs}\}.$$

A most general reason for the failure would be a maximal element of this set. Note that a "reason" here is a value-set assignment, not a complete valuation, nor a constraint. This leads to a specification of conflict analysis as a problem: given a value-set assignment vs that fails some constraint $C_n \in Cs$, find a value-set assignment ws that

1. refines to countermodels: $\gamma(ws) \subseteq \mathit{cmod}_{Cs}$
2. generalizes vs: $\gamma(vs) \subseteq \gamma(ws)$, or $ws \;\sqsupseteq\; vs$
3. is as general as possible: find a maximal element satisfying (1) and (2).

D'Silva, Haller and Kroening (DHK) [2] represent $\mathit{Reasons}(vs, C_n)$ as a Galois Connection that under-approximates the countermodels of Cs, dually to how the value-set domain itself is an over-approximating domain for models of Cs. The concrete operation of interest here is $\mathit{cmod}_{Cs}(X)$, and, roughly speaking, a sound (under)approximation of it is iterated to find a maximal reason for the failure. Once such a reason is found, it is converted to a constraint that is used both for backjumping and for learning.

While there is a nice conceptual symmetry here (by casting both propagation inference and conflict analysis as abstract interpretations over/under the same concrete domain), there are several problems with this approach. First, in SAT, there is a one-to-one correspondence between partial assignments and constraints, so a most-general partial assignment can be directly converted to a constraint. It is not obvious that there will be such a bijection in other problem domains. Second, the iteration is not focused on the particular constraint that fails at the failure point. Third, the crucial discovery that one can perform constraint analysis by resolving from the failure point is glossed over in the DHK approach: constraint resolution to the 1-UIP [7, 13] is treated as a "heuristic" that serves to create a starting point for an iterative clause minimization process [12].

Conflict Analysis as guided inference

We believe that an alternative approach provides a more effective and satisfying formulation. This approach focuses on constraints directly rather than value-set assignments. By analogy to the DHK approach, we provide an alternative formulation of the reasons for a failure:

$$\mathit{Reasons}(vs, C_n) \;=\; \{C \mid (Cs \vdash C) \;\wedge\; (vs \models C = \texttt{False})\}.$$

A most general reason for the failure would be a strongest constraint in this set. This leads to an alternative specification of conflict analysis as a problem: given a value-set assignment vs that fails some constraint $C_n \in Cs$, find a constraint C that

1. is a consequence of Cs: $Cs \vdash C$
2. generalizes the failure of vs: $vs \models C = \texttt{False}$
3. is as strong as possible: find a minimal element satisfying (1) and (2) (using implication as a partial order).

A direct realization of this specification can be formalized in terms of iterated constraint resolution along the chain of propation inferences made at the current depth, and including a form of constraint normalization that generalizes [12]. We discuss next the particular mathematical results that justify and inform this approach.

Resolution of constraints in arbitrary logics

State of the art solvers make use of constraint resolution to infer at runtime new constraints and record them as learned constraints. Since the constraint theory presented above is independent of logic, the operation of resolution can only be specified abstractly, not defined. When details of the constraint logic are specified concretely, then we can synthesize or manually develop the necessary resolution operation. In this section, we lay out the abstract theory of constraint resolution and give a variety of examples of its application.

Definition. If constraints C_1 and C_2 each contain an occurrence of variable x, then a *resolvant* of C_1 and C_2 is a quantifier-free formula that is equivalent to $\exists(x)\big(C_1 \ \wedge \ C_2\big)$.

We speak of "the" resolvant of two constraints - there is only one resolvant from a semantic point of view, whereas there can be many syntactic expressions of it. Hence we specify the Resolve operator, but expect that it can have many implementations.

For particular problems it is clear how to perform resolution. For example, in SAT and general CSP problems, one uses the cut rule from propositional logic; for time-window-based scheduling we need generalized resolution with special relations [6], and for variants of ILP we use cutting planes (essentially Fourier-Motzkin). The challenge is to unify these examples and see how to express a generalized resolution rule that applies at the abstract level of Global Search theory. One consideration is that the effect of resolving the pruning condition and the last decision is that we want to eliminate the last decision, so that the inferred formula is a pruning condition at the previous step. Thus eliminating a variable seems to be an essential ingredient.

Given two formulas $\mathcal{F}[a]$ and $\mathcal{G}[a]$ containing a common subexpression a, we desire to find a consequence that eliminates a. We do not specify in which logic these formulas are expressed; the point is to characterize the resolution inference rule in an arbitrary logic.

Proposition 3.1 $\mathcal{F}[a] \ \wedge \ \mathcal{G}[a] \implies \exists(a)\big(\mathcal{F}[a] \ \wedge \ \mathcal{G}[a]\big)$.

Proof: In any model for the left-hand side there exists some valuation for the subexpression a. $\square$

We conjecture that the resolution inference rule can always be calculated by applying quantifier elimination to the right-hand side in Proposition 3.1 in the theory of the problem domain. Quantifier elimination may only apply under certain conditions. For example, the resolution rule for propositional and first-order logic requires that the expression to be eliminated has opposite polarity in the two expressions. We conjecture that the generic precondition on Resolve is that the two constraints can come into conflict in the following sense: there exists a partial valuation under which the feasible values some variable (a in the proposition) are disjoint:

$$\exists vs(\{a|a \in vs(a) \ \wedge \ \mathcal{F}[a]\} \cap \{a|a \in vs(a) \ \wedge \ \mathcal{G}[a]\} = \{\}.$$

The Proposition justifies the following specification of the resolution operation in an arbitrary logic.

weakResolve is a variant in which the resolvant is a weakening of $\exists(a)\big(\mathcal{F}[a] \ \wedge \ \mathcal{G}[a]\big)$ for logics in which the resolvant itself is not expressible (i.e. the logic is not closed under strong resolution).

```
op Resolve(C1:Constraint)(C2:Constraint)
          (v:Variable|  v in? (varsOf C1) && v in? (varsOf C2)):
          {C:Constraint | Equivalent C (mkExistential v (mkConjunction C1 C2))}

theorem resolution_inference is
  fa (C1:Constraint,C2:Constraint)
     Entails(mkConjunction(C1,C2), Resolve(C1,C2))
```

We show next several examples of how we can calculate resolution rules for various logics.

Example 1. Propositional Logic's Cut Rule: $A \vee B,\ \neg B \vee C \vdash \ A \vee C$.

We calculate as follows:

$(A \vee B) \ \wedge \ (\neg B \vee C)$

$\Longrightarrow \qquad \{$ Proposition 3.1 $\}$

$\qquad \exists(B)\big((A \vee B) \ \wedge \ (\neg B \vee C)\big)$

$= \qquad \{$ unfolding the existential; or case analysis $\}$

$\qquad ((A \vee \mathit{false}) \ \wedge \ (\neg \mathit{false} \vee C)) \ \vee \ ((A \vee \mathit{true}) \ \wedge \ (\neg \mathit{true} \vee C))$

$=$ $\quad\{$ Simplifying $\}$

$\quad (A \wedge \mathit{true}) \vee (\mathit{true} \wedge C)$

$=$ $\quad\{$ Simplifying$\}$

$\quad A \vee C.$

Note that in the cut rule, the occurences of the eliminated B proposition have postive and negative polarity respectively. This insight was generalized by Manna and Waldinger who developed a generalized resolution calculus for deductive synthesis [5] and extended it to handle special relations [6].

Example 2. Generalized Resolution in First-order Predicate Calculus: If a occurs positively in $\mathcal{F}[a]$ and negatively in $\mathcal{G}[a]$, then $\mathcal{F}[a] \wedge \mathcal{G}[a] \vdash \mathcal{F}[\mathit{false}] \vee \mathcal{G}[\mathit{true}]$.

Lemma 3.1 *Let $H[a]$ be monotone (antitone) in $a : Boolean$. If $H[a]$ holds in some structure, then $H[\mathit{true}]$ (resp. $H[\mathit{false}]$) also holds.*

Proof: $H[a]$ iff $\exists(b)\big(H[b]\big)$ iff $H[\mathit{true}]$, where the last step holds by QE. Similar reasoning applies to the antitone case.

First, assume we have a model of $\mathcal{F}[a] \wedge \mathcal{G}[a]$. By Lemma 3.1 we also then have $\mathcal{F}[\mathit{true}] = \mathit{true} \wedge \mathcal{G}[\mathit{false}] = \mathit{true}$.

We calculate as follows:

Assume: $\mathcal{F}[a] \wedge \mathcal{G}[a]$
$\quad\wedge\, \mathcal{F}[\mathit{true}] = \mathit{true} \wedge \mathcal{G}[\mathit{false}] = \mathit{true}$
$\quad\wedge\, \mathcal{F}[\mathit{false}] \implies \mathcal{F}[\mathit{true}]$
$\quad\wedge\, \mathcal{G}[\mathit{true}] \implies \mathcal{G}[\mathit{false}]$
Simplify: $\mathcal{F}[a] \wedge \mathcal{G}[a]$

$\mathcal{F}[a] \wedge \mathcal{G}[a]$

$\implies$ $\quad\{$ Proposition 1 $\}$

$\quad \exists(a)\big(\mathcal{F}[a] \wedge \mathcal{G}[a]\big)$

$=$ $\quad\{$ unfolding the existential; or case analysis $\}$

$\quad (\mathcal{F}[\mathit{false}] \wedge \mathcal{G}[\mathit{false}]) \vee (\mathcal{F}[\mathit{true}] \wedge \mathcal{G}[\mathit{true}])$

$=$ $\quad\{$ Simplifying using Lemma 1 $\}$

$(\mathcal{F}[\mathit{false}] \ \wedge \ \mathit{true}) \ \vee \ (\mathit{true} \ \wedge \ \mathcal{G}[\mathit{true}])$

$=$ { Simplifying}

$\mathcal{F}[\mathit{false}] \ \vee \ \mathcal{G}[\mathit{true}]$.

Example 3: 0,1-Pseudo-Boolean Linear (PBL)

In this case we have linear inequalities over 0,1 variables with positive coefficients and bounds.

let $C1 = \Sigma a_i \cdot x_i \geq d_1$ and $C2 = \Sigma b_i \cdot x_i \geq d_2$ where i ranges over $\{1, .., n\}$.

$C1 \oplus C2$

$=$ { assume a_1 is positive and b_1 negative, then we can resolve on x_1, yielding by definition }

$\exists x_1 (\Sigma a_i \cdot x_i \geq d_1 \ \wedge \ \Sigma b_i \cdot x_i \geq d_2)$

$=$ { let $\Sigma_1 = \Sigma_{i \geq 2} a_i \cdot x_i$ and $\Sigma_2 = \Sigma_{i \geq 2} b_i \cdot x_i$ }

$\exists x_1 (a_1 \cdot x_1 + \Sigma_1 \geq d_1 \ \wedge \ b_1 \cdot x_1 + \Sigma_2 \geq d_2)$

$=$ { isolating x_1 in both conjuncts, noting negative b_1 }

$\exists x_1 (x_1 \geq (d_1 - \Sigma_1)/a_1 \ \wedge \ x_1 (d_2 - \Sigma_2)/b_1)$

$=$ { QE aka Fourier-Motzkin }

$(d_2 - \Sigma_2)/b_1 \geq (d_1 - \Sigma_1)/a_1$

$=$ { rearranging and simplifying}

$\Sigma_{i \geq 2} (a_i/a_1 - a_i/b_1) \cdot x_i \geq (d_1/a_1 d_2/b_1)$.

Resolution inferences in ILP are sometimes called (Gomory) cutting planes.

We can establish several results about the resolve function. We use the circle-plus sign as an infix notation for resolve: $\mathit{Resolve}(C_1, C_2) = C_1 \oplus C_2$.

Proposition 3.2 *1.* $\mathit{mod}_{C_1}(X) \cap \mathit{mod}_{C_2}(X) \ \subseteq \ \mathit{mod}_{C_1 \oplus C_2}(X)$
2. $\mathit{cmod}_{C_1}(X) \cup \mathit{cmod}_{C_2}(X) \ \supseteq \ \mathit{cmod}_{C_1 \oplus C_2}(X)$

Proof: We show (1), and then (2) follows by duality.

$$
\begin{array}{lll}
\multicolumn{2}{l}{mod_{C_1}(X) \cap mod_{C_2}} & \\
 = & \{m \mid m \in X \ \wedge \ m \models C1\} \cap \{m \mid m \in X \ \wedge \ m \models C2\} & \text{by definition} \\
 = & \{m \mid m \in X \ \wedge \ m \models C1 \ \wedge \ m \models C2\} & \\
 \subseteq & \{m \mid m \in X \ \wedge \ m \models C1 \oplus C2\} & \text{Proposition 2.2} \\
 = & mod_{C_1 \oplus C_2}(X). & \text{by definition}
\end{array}
$$

Proposition 3.3 *Let x be a variable over a lattice-structured domain. Let C_1 be a constraint that is monotone or antitone in x. Similarly for constraint C_2. Let Θ be a valuation over $Vars(C_1) \bigcup Vars(C_2) \backslash \{x\}$ such that C_1 and C_2 are satisfiable in Θ and*

$$\Theta \models \{x \mid C_1\} \bigcap \{x \mid C_2\} = \{\}$$

then (1) x has opposite polarity in C_1 and C_2 and (2) the resolvant of C_1 and C_2 is false in Θ.

Proof: Assume that x is monotone both C_1 and C_2. Let x_1 be an element of $\{x \mid C_1\Theta\}$ and let x_2 be an element of $\{x \mid C_2\Theta\}$. Since $x_1 \leq x_1 \sqcup x_2$ and $x_2 \leq x_1 \sqcup x_2$, then by the monotonicity assumption, we also have

$$C_1[x_1] \implies C_1[x_1 \sqcup x_2]$$

and

$$C_2[x_2] \implies C_2[x_1 \sqcup x_2].$$

Therefore $x_1 \sqcup x_2$ satisfies both C_1 and C_2, contradicting our assumption. A similar argument applies in the case that x is antitone both C_1 and C_2, except that we consider the element $x_1 \sqcap x_2$.

To show part (2), the assumption that

$$\Theta \models \{x \mid C_1\} \bigcap \{x \mid C_2\} = \{\}$$

is equivalent to

$$\Theta \models \neg\exists(x)\big(C_1 \ \wedge \ C_2\big);$$

i.e. the resolvant of C_1 and C_2 is false in Θ. $\Box$

The straightforward interpretation of the Proposition is that the resolvant of two constraints can be used to preclude some candidate valuations from the model-level search.

Notation: if vs is a partial/value-set valuation, then $vs(x)$ is the set of values that variable x can be assigned. A partial valuation is effectively complete if $vs(x)$ is a singleton for each variable x. When C is a constraint, then $vs[C]$ is `False` if

$$\forall(m : Valuation)(m \in \gamma(vs) => \neg m \models C)$$

i.e. C fails all extensions of vs, and $vs[C]$ is `unknown` otherwise. Let

$$vs[C](x) = \{v \mid \exists(m : Valuation)(m(x) = v \ \wedge \ m \models C \ \wedge \ m \in \gamma(vs)\}.$$

Proposition 3.4 *In the GS-CDBL scheme, if we use Arc Consistency for propagation, and resolve backwards from a failure point, then each resolvant back to the decision variable will be false under the current partial valuation.*

Proof: Assume that we use a value-set abstract domain, and propagation reduces the value sets using the definite constraints derived from Arc Consistency. We proceed by induction from the failure point at the current depth d. We examine the sequence of inferences that follow from the decision at depth d. Suppose that the last propagation inference reduced the value set of variable x, so $vs_n(x) \subset vs_{n-1}(x)$. Furthermore, suppose that the failed constraint is C_n, so $vs_n[C_n] = \texttt{False}$. Generally, let the i^{th} inference step at depth d use constraint C_i, and let $\hat{C}_i$ be the resolvant of $\hat{C}_{i+1}$ and C_i, where $\hat{C}_n = C_n$ initially. We need to show that if $\hat{C}_{i+1}$ fails in the current partial valuation, then so does $\hat{C}_i$. We reason as follows:

$vs_i[\hat{C}_i]$

$=$ $\{$ definition of resolvant $\}$

$vs_i[\exists(x_i)(C_i(x_i) \ \wedge \ \hat{C}_{i+1}(x_i))]$

$=$ $\{$ Simplifying $\}$

$\exists(x_i)(vs_i[C_i(x_i)] \ \wedge \ vs_i[\hat{C}_{i+1}(x_i)])$

$=$ $\{$ Simplifying $\}$

$\exists(x_i, v_i)(v_i \in vs_i(x_i) \ \wedge \ v_i \in vs_i[C_i](x_i) \ \wedge \ v_i \in vs_i[\hat{C}_{i+1}](x_i))$

$=$ $\{$ By assumption that $\hat{C}_{i+1}$ fails in vs, the set $vs_i[\hat{C}_{i+1}](x_i)$ must be disjoint from $vs_i[C_i](x_i)$ $\}$

$\exists(v_i)\texttt{False}$

$=$ $\{$ Simplifying $\}$

$\texttt{False}$.

This result is important because in conflict analysis we wish to infer a constraint that fails in the current partial valuation that can be used to perform propagation inferences at an earlier depth. The proposition asserts that simply resolving backwards from a failure point towards the most recent decision will yield such failing constraints.

4 Related Work

It is useful to step back and reflect on the history of SAT solving. The first SAT solving method was by Davis-Putnam in 1960 and was a pure proof search via the cut rule (which is resolution on propositional formulas). This algorithm is complete but very inefficient. The main reason for the inefficiency is our inability to control the inferences to be relevant. The proof space is vast and contains many valid inferences that are irrelevant to the problem at hand. This is a fundamental weakness of proof-search methods in most expressive logics. Hence the focus on logics that have efficient decision procedures - algorithms that calculate proofs without having to search.

The next step, soon afterward, was by Davis-Logemann-Loveland in 1962, which based the algorithm on search in model space – a model of the given constraints is built up an assignment at a time. Inference was restricted to a special case of the cut rule: unit resolution, which forces an assignment to a variable. This proved to be a much more efficient algorithm. Note how the Davis-Logemann-Loveland algorithm combines model search and inference. Inference is tightly controlled in such a way as to be much more relevant - it is only applied in contexts such that it directly contributes to extending the search in model space. The model search is driving the solver and inference is there to help the model search rather than to build a proof directly.

Research into backjumping and learning in SAT starting in the 70's culminated in the dependency-directed backtracking and learning mechanisms in GRASP from Marques-Silva and Sakallah in the 90's [7], with several significant advances by Zhang et al. at Princeton in the (z)CHAFF system[9]. Inference (again the cut rule) is applied not only to further the model search, but also to infer how to proceed when model search reaches a deadend, a contradiction. Again inference is used in a tightly controlled way to foster the model search rather than to construct a proof.

These reflections on SAT solving help to guide our thinking about synthesizing best-practice constraint solvers. They provide some insight into the development of SMT solvers, which are working through the issues of how to combine the model search of SAT with efficient theory-inference. Opportunities still exist to apply these insights into optimization problems - how can inference help guide the model-search towards optimal cost models?

5 Concluding Remarks and Open Issues

The main contribution of this paper has been to lay out a standard pattern for the derivation of correct-by-construction CSP solvers using the conflict-directed backjumping and learning paradigm. Our emphasis has been on formalizing a design theory for this class of solvers, which is necessarily independent of the particulars of the constraint logic.

There are several issues that remain open and are subjects for future research. We made progress on developing a native Metaslang calculator for use in transformations, but more needs to be done, e.g. to support the calculation of a classification morphism from the `GS_CDBL` to a given problem specification. The older KIDS system [11] had a modified first-order prover that could generate refinements of this kind.

Another issue arises from the distinction between problem constraints that are bound at specification time and those that are bound at runtime (as in SAT and most mathematical programming problems). The calculations for generating propagation code depend on the form of the constraints. Our current solution to this problem is to have the user perform the definite constraint calculations for example constraints and then to specify the general pattern to implement propagation.

References

[1] Dechter, R. *Constraint Processing.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[2] D'Silva, V., Haller, L., and Kroening, D. Satisfiability solvers are static analysers. In *SAS* (2012), pp. 317–333.

[3] D'Silva, V., Haller, L., and Kroening, D. Abstract conflict driven learning. In *POPL* (2013), pp. 143–154.

[4] Kestrel Institute. *Specware System and documentation*, 2003. http://www.specware.org/.

[5] Manna, Z., and Waldinger, R. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems 2*, 1 (January 1980), 90–121.

[6] Manna, Z., and Waldinger, R. Special relations in automated deduction. In *Automata, Languages and Programming* (1985), Springer LNCS 194, pp. 413–423.

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

[8] McMillan, K. L., Kuehlmann, A., and Sagiv, M. Generalizing dpll to richer logics. In *CAV'09* (2009), pp. 462–476.

[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] Rehof, J., and Mogensen, T. Tractable constraints in finite semilattices. In *Science of Computer Programming* (1996), Springer-Verlag, pp. 285–300.

[11] 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.

[12] Sörensson, N., and Biere, A. Minimizing learned clauses. In *SAT* (2009), pp. 237–243.

[13] 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 $F : \langle A, \preceq\rangle \to \langle Boolean, \wedge, \vee, \Rightarrow\rangle$			
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 $F : \langle A, \preceq\rangle \to \langle Boolean, \wedge, \vee, \Rightarrow\rangle$			
-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 $F : \langle Boolean, \Rightarrow\rangle \to \langle Boolean, \wedge, \vee, \Rightarrow\rangle$			
3.1	$\exists(a : Boolean)\ F(a) \ = \ F(true)$	$\forall(a : Boolean)\ F(a) \ = \ F(false)$	3.2

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

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

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

Resolution in Predicate Calculus

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

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