Noname manuscript No.
(will be inserted by the editor)

Generalized Support and Formal Development of Constraint Propagators

James Caldwell · Ian P. Gent · Peter Nightingale

the date of receipt and acceptance should be inserted later

Abstract The concept of *support* is pervasive in constraint programming. Traditionally, when a domain value ceases to have support, it may be removed because it takes part in no solutions. Arc-consistency algorithms such as AC2001 [8] make use of support in the form of a single domain value. GAC algorithms such as GAC-Schema [7] use a tuple of values to support each literal. We generalize these notions of support in two ways. First, we allow a set of tuples to act as support. Second, the supported object is generalized from a set of literals (GAC-Schema) to an entire constraint or any part of it.

We design a methodology for developing correct propagators using generalized support. A constraint is expressed as a family of support properties, which may be proven correct against the formal semantics of the constraint. Using Curry-Howard isomorphism to interpret constructive proofs as programs, we show how to derive correct propagators from the constructive proofs of the support properties. The framework is carefully designed to allow efficient algorithms to be produced. Derived algorithms may make use of *dynamic literal triggers* or *watched literals* [15] for efficiency. Finally, two case studies of deriving efficient algorithms are given.

1 Introduction

In this paper we provide a formal development of the notion of support in constraint satisfaction. This notion is ubiquitous and plays a vital role in the understanding, development, and implementation of constraint propagators, which in turn are the keystone of a successful constraint solver. While we focus on a formal

James Caldwell
Department of Computer Science, University of Wyoming, 1000 E. University Ave., Laramie, WY 82071-3315, USA
E-mail: jlc@uwyo.edu

Ian P Gent · Peter Nightingale
School of Computer Science, Jack Cole building, University of St Andrews, St Andrews, Fife KY16 9SX, UK
E-mail: {ian.gent,pwn1}@st-andrews.ac.uk

development in this paper, our purpose is not to describe formally what is currently seen in constraint satisfaction. Instead, we generalize the notion of support so that it can be used in a wider variety of propagators. The result is the first step in a twin programme of developing a formal understanding of constraint algorithms, while also developing notions such as generalized support which should lead to improved constraint algorithms in the future.

The methodology presented here for formal development of propagators is based on the proofs-as-programs and propositions-as-types interpretations of constructive type theory [11,17]. Like the earlier development in [9], the approach presented here uses a constructive type theory as the formal framework for specifying and developing programs. There, the proofs were mechanically checked in the Nuprl theorem prover [12], here the development is formal but proofs have not been mechanically checked.

1.1 Overview of the Constraint Satisfaction Problem

A constraint is simply a relation over a set of variables. Many different kinds of information can be represented with constraints. The following are simple examples: one variable is less than another; a set of variables must take distinct values; task A must be scheduled before task B; two objects may not occupy the same space. It is this flexibility which allows constraints to be applied to many theoretical, industrial and mathematical problems.

The classical constraint satisfaction problem (CSP) has a finite set of variables, each with a finite domain, and a set of constraints over those variables. A solution to an instance of CSP is an assignment to each variable, such that all constraints are simultaneously satisfied — that is, they are all true under the assignment. Solvers typically find one or all solutions, or prove there are no solutions. The decision problem ('does there exist a solution?') is NP-complete [1], therefore there is no known polynomial-time procedure to find a solution.

1.2 Solving CSP

Constraint programming includes a great variety of domain specific and general techniques for solving systems of constraints. Since CSP is NP-complete, most algorithms are based on a search which potentially explores an exponential number of nodes. The most common technique is to interleave splitting and propagation. Splitting is the basic operation of search, and propagation simplifies the CSP instance. Apt views the solution process as the repeated transformation of the CSP until a solution state is reached [1]. In this view, both splitting and propagation are transformations, where propagation simplifies the CSP by removing domain values which cannot take part in any solution. A splitting operation transforms a CSP instance into two or more simpler CSP instances, and by recursive application of splitting any CSP can be solved.

Systems such as Choco [21], ILOG Solver [20] and Minion [14,15] implement highly optimized constraint solvers based on search and propagation, and (depending on the formulation) are able to solve extremely large problem instances quickly.

Our focus in this paper is on propagation algorithms. A propagation algorithm operates on a single constraint, simplifying the containing CSP instance by removing values from variables in the scope of the constraint. Values which cannot take part in any solution are removed. For example, a propagator for $x \leq y$ might remove all values of x which are greater than the largest value of y. Typically propagation algorithms are executed iteratively until none can make any further simplifications.

1.3 Proofs to propagators

Researchers frequently invent new algorithms and (sometimes) give proofs of correctness, of varying rigour. In this paper we provide a formal semantics of CSP. This allows us to formally characterize correctness of constraint propagators, and therefore aid the proof of correctness of propagators. Following this, we lay the groundwork for automatic generation of correct propagators. The method is to write a set of *support properties* which together characterize the constraint. Each property is inserted into a schema, and a constructive proof of the schema is generated. This proof is then translated into a correct-by-construction propagator. This method is based on the concept of *generalized support*, described in the next section. Finally, we give examples of this method by deriving propagators for the `element`, `occurrenceleq` and `occurrencegeq` constraints.

1.4 Generalized support

Central to this work is the notion of support. This notion is used informally in many places (for example, in the description of the algorithm GAC-Schema [7]) and more formally by Bessière [5]. We generalize the concept of support, and develop a formal framework to allow us to produce rigorous proofs of the correctness of propagators that exploit the generalized concept of support.

Support is a natural concept in constraint programming. Constraint propagators remove unsupported values from variable domains, thus simplifying a CSP instance. Supported values cannot be removed, since they may be contained in a solution. Thus a support is evidence that a value (or set of values) may be contained in a solution. If no support exists, it is guaranteed that a value (or set of values) is not contained in any solution.

A *support property* characterises the supports of a particular value (or set of values) for a particular constraint. For example, three support properties of an element constraint are given by Gent et al. [15]. Each of these three properties is used to create a propagator, such that the three propagators together achieve generalized arc consistency. In this instance, writing down support properties assisted in proving the propagators correct.

We show that correct support properties can be used to create propagators that are correct by construction. We describe a general "propagation schema", which is a description of what should be proved when support is lost for a given support property. This captures how propagators work in practice. They are "triggered" when it is noted that the current support is lost. The propagator then seeks to re-establish support. This might be possible on the current domains, or it may need to narrow domains (i.e. remove some values of some variables), or it may be that no new support is possible and the constraint is guaranteed to be false. The propagation schema specialised for a given support property can be proven constructively. The proof contains sufficient information to be translated into a correct propagator. We envisage two main uses for such a propagator. For some constraints, it may be an efficient propagator that can be used directly. Otherwise, the constructed propagator may be used as part of an informal argument for the correctness of an efficient propagator.

1.5 Related Work

There are a number of items of related work with related or similar goals, however the approach taken in each case is quite different to our approach. Apt and Monfroy [2] generate propagation rules such as $X = s \to y \neq a$, where X is a vector of CSP variables, s is a vector of values within the initial domain of X, y is a CSP variable and a is a value in the initial domain of y. Rules correspond directly to propagation in a constraint solver (*ie* when X is assigned s, a is removed from the domain of y). A set of rules is generated for a given constraint by a search over the (potentially very large) space of possible rules. In contrast, our approach is much broader in that it is not restricted to generating implication rules. Our framework allows both the derivation of new propagators and proof of correctness of existing ones.

Beldiceanu, Carlsson and Petit [4] describe constraints using finite state automata extended with counters. For a constraint C, the automaton for C can check whether any given assignment satisfies C. Beldiceanu, Carlsson and Petit give a method to translate an automaton into a set of short constraints (a decomposition) such that propagating them will propagate the original constraint C, and there are (in some cases) guarantees of the strength of propagation. The approach has been subsequently refined, for example by linking overlapping prefixes and suffixes of constraints [3]. Their approach generates decompositions of a particular form, whereas in this paper our focus is on deriving efficient propagators.

Cohen, Jefferson and Petrie [10] studied the properties of triggers, in particular comparing static triggers with movable triggers on a number of constraint classes and consistencies. They demonstrate that movable triggers can lead to much more efficient propagators. To do this they generalise the concept of support in a similar way to us, however their work treats each propagator as a monolithic black box whereas we are interested in constructing propagators and proving correctness and other properties of them.

2 Definitions and Notation

2.1 The Standard Mathematical Account

We start by giving the standard definition of a constraint satisfaction problem (*e.g.* see [13, 5]). Formal definitions of the notations used here are given below.

Definition 1 (Constraint Satisfaction Problem) A *Constraint Satisfaction Problem* (CSP) is given by a triple $\langle X, \sigma, C \rangle$ where X is a k-tuple of variables $X =$

$\langle x_1, \cdots, x_k\rangle$ and σ is a *signature* (a function $\sigma : X \to 2^{\mathbb{Z}}$ mapping variables in X to their corresponding domains, *i.e.* such that $\sigma(x_i) \subseteq \mathbb{Z}$ is the finite domain of variable x_i.) C is a tuple of extensional constraints $C = \langle C_1, \cdots, C_m\rangle$ where each C_i is of the form $\langle Y, R_Y\rangle$ where $Y \subseteq X$ is a tuple of variables called the *schema* or *scope* of the constraint C_i. Also, R_Y is a relation given by a subset of the Cartesian products of the domains of the variables in the scope Y and is called the *extension* of C_i.

Definition 2 (Satisfying tuple) We say a Z-tuple τ *satisfies* constraint $\langle Y, R_Y\rangle$ if $Y \subseteq Z$, and the projection $Y[\tau]$ is in R_Y (*i.e.* if the projection of the scope Y from τ is in R_Y.)

Definition 3 (Solution) A *solution* to a CSP $\langle X, \sigma, C\rangle$ is a tuple τ, with schema X, such that τ satisfies every constraint in C.

2.2 Variable Naming Conventions, Ranges, and Literals

We use lower case letters (possibly subscripted or primed) from near the end of the Latin alphabet $\{w, x, y, z\}$ to denote variables. We use Latin letters $\{i, j, k\}$ to denote integer indexes, and use the Latin letters occurring early in the alphabet $\{a, b, c, d\}$ (possibly subscripted) to denote arbitrary integer values.

$$\{b..c\} \overset{def}{=} \{a \in \mathbb{Z} \mid b \leq a \wedge a \leq c\}$$

We write 2^A to denote the powerset (set of all subsets) of A. A *literal* is a variable-value pair (*e.g.* $\langle x, 5\rangle$).

2.3 Vectors

We use uppercase letters $W, X, Y, Z, ...$ to denote vectors of variables. We use the Greek letters $\{\tau, \tau', \tau_1, \tau_2 \cdots\}$ to denote tuples of integer values.

We write finite vectors as sequences of values enclosed in angled brackets, (*e.g.* $\langle x, y, z\rangle$). The empty vector is written $\langle\rangle$. We take the operation of prepending a single element to the left end of a vector as primitive and denote this operation $x \cdot Y$. We abuse this notation by writing $X \cdot Y$ for the concatenation of vectors X and Y. We write $|Y|$ to denote the length of vector Y. Given a vector Y, we write $Y[i]$ to denote the (zero-based) i^{th} element of Y. This operation is undefined if $i \notin \{0..|Y| - 1\}$.

Membership in a vector;

$$z \in Y \overset{def}{=} \exists i : \{0..|Y| - 1\}. Y[i] = z$$

We will sometimes need to collect the set of indexes to an element in a vector. $Y[[z]] \overset{def}{=} \{i \in \{0..|Y| - 1\} \mid Y[i] = z\}$. Thus, $\langle x, y, z, x\rangle[[x]] = \{0, 3\}$. Note that $Y[[z]] \neq \emptyset$ iff $z \in Y$ and also each index in $Y[[z]]$ is a witness for $z \in Y$.

If $y \in Y$, we write $Y - y$ to denote the vector obtained from Y by deleting the leftmost occurrence of y from Y. $Y - y = Y$ if $y \notin Y$. We write $Z - Y$ for the vector obtained by removing leftmost occurrences of all $(y \in Y)$ from Z. Given a vector Z, we write $\{Z\}$ to denote the set of values in Z and given a set of variables S we write $\langle S\rangle$ to denote a vector of the variables in S; the reader may assume the variable in $\langle S\rangle$ occur in increasing lexicographic order. Intersection and unions are defined on vectors by taking them as sets: $X \cap Y \overset{def}{=} \langle\{X\} \cap \{Y\}\rangle$; and $X \cup Y \overset{def}{=} \langle\{X\} \cup \{Y\}\rangle$. We write $Y \subseteq X$ to mean $\{Y\} \subseteq \{X\}$, *i.e.* that every element in Y is in X with no stipulations on relative lengths of X or Y or on the order of their elements.

2.4 Signatures

A signature σ is a function mapping variables in X to their associated domains. Thus, signatures are functions $\sigma : X \to 2^{\mathbb{Z}}$ where in practice, the subset of integers mapped to is finite. If σ and σ' are signatures mapping variables in X to their finite integer domains:

$$\sigma' \sqsubseteq_X \sigma \overset{def}{=} \forall x \in X.\sigma'(x) \subseteq \sigma(x)$$

We write $\sigma' \sqsubset_X \sigma$ if $\sigma' \sqsubseteq_X \sigma$ and $\exists x \in X : \sigma'(x) \sqsubset \sigma(x)$, *i.e.* if some domain of σ' is a proper subset of the corresponding domain of σ. We drop the schema subscript when the schema is clear from the context. We state the following without proof.

Lemma 1 (Signature Inclusion Well-founded) *The relation $\sqsubset$ is well-founded if restricted to signatures with finite domains.*

2.5 Relations

In the description of a CSP given above, a constraint $\langle Y, R_Y\rangle$ is a relation where the schema Y gives the attribute names and R_Y is the set of tuples in the relation.

Given a signature σ mapping variables in schema Y to their domains, a relation $\langle Y, R_Y\rangle$ is *well-formed* with respect to σ iff the following conditions hold:

i. All tuples in R_Y have length $|Y|$
ii. The values in each column come from the specified domain for that column.

$$\forall \tau : R_Y.\, \forall i : \{0..|Y| - 1\}.\, \tau[i] \in \sigma(Y[i])$$

Schemata are vectors of variable names with no restriction on how many times a variable may occur. Thus it is possible to have a wellformed relation whose schema has common names for multiple columns. Given a signature σ over a schema X, a tuple τ is called a X-tuple if $\langle X, \{\tau\}\rangle$ is well-formed w.r.t. σ. In this case, we write $X\text{–}tuple_\sigma(\tau)$. We write $X\text{–}tuple_\sigma$ for the set of tuples satisfying this condition.

2.5.1 Tuple Coherency

Conceptually, relations provide a representation for storing valuations (assignments of values to variables) and so we must distinguish between tuples which represent coherent valuations (even when their schemata may contain duplicate variable names) and tuples that do not. This motivates the following definitions.

The wellformedness condition on relations requires values in columns labeled by a variable come from the domain of that variable, but does not rule out cases where a single tuple with multiple columns named by the same variable have different values in those columns.

Example 1 Consider the relation

$$\langle\langle x, x, y\rangle, \{\langle 1,2,3\rangle, \langle 1,1,3\rangle, \langle 2,2,3\rangle\}\rangle$$

The variable x occurs twice in the schema and the first tuple in the schema assigns different values to x, this tuple is not coherent.

An X-tuple τ is *coherent w.r.t. variable z* iff the following holds.

$$\mathbf{coh}\{X, z\}(\tau) \overset{def}{=} \forall i, j : X[[z]].\ \ \tau[i] = \tau[j]$$

We say a tuple is *incoherent w.r.t. z* if it is not coherent. Note that this definition is sensible whether $z \in X$ or not. A simple consequence of the definition is that an X-tuple τ is incoherent w.r.t. variable z iff

$$\exists i, j : X[[z]].\ \ \tau[i] \neq \tau[j]$$

An X-tuple τ is *coherent with schema Y* iff it is coherent w.r.t. all variables $z \in Y$.

$$\mathbf{coh}\{X, Y\}(\tau) \overset{def}{=} \forall z \in Y.\ \mathbf{coh}\{X, z\}(\tau)$$

We say an X-tuple is *incoherent with respect to schema Y* if it is not coherent w.r.t. Y. Only coherent tuples count as solutions (Def. 3).

Remark 1 In many constraint solvers, incoherent tuples may arise during a computation, but they are never counted among solutions. For example, the Global Cardinality constraint $\texttt{GCC}(\langle x, x, y\rangle, \langle 1, 2\rangle, \langle (2 \ldots 3), (1 \ldots 2)\rangle)$ (stating that value 1 occurs two or three times, and value 2 occurs once or twice among variables $\langle x, x, y\rangle$) could generate the incoherent tuple $\langle 1, 2, 1\rangle$ internally when using Règin's algorithm [23].[1] Generating incoherent tuples affects both the internal state of a constraint propagator, and the number of vertices in the search tree.

Strictly speaking, because incoherent tuples do not count as solutions, the semantics could be specified simply disallowing them. However, this approach would rule out faithful finer grained representations of the internal states of constraint solvers which do generate incoherent tuples *e.g.* when searching for support. Based on this, we have decided to include them although this adds some complexity to the specification.

[1] Règin's algorithm [23] is polynomial-time and enforces GAC iff the schema contains no duplicate variables. With duplicate variables, enforcing GAC on GCC is NP-Hard [6], therefore it is sensible to use Règin's algorithm in this case even though it will not enforce GAC.

2.5.2 Selection

Selection is an operation mapping relations to relations generating new ones from old by filtering rows (tuples) based on predicates on the values in the tuple.

Given a relation $\langle Y, R_Y \rangle$ and an index $i \in \{0..|Y|-1\}$, and a value (say a), *index selection* is defined as follows.

$$select_{(i=a)}(R_Y) \overset{def}{=} \{\tau \in R_Y | \tau[i] = a\}$$

The tuples selected from a relation by index selection are not guaranteed to be coherent with respect to schema Y.

Given a relation $\langle Y, R_Y \rangle$ and a variable x and a value (say a), *value selection* is defined as follows.

$$select_{(x=a)}(R_Y) \overset{def}{=} \{\tau \in R_Y | \forall i : Y[[x]].\ \tau[i] = a\}$$

Thus a tuple τ is included in a selection $select_{x=a}R_Y$ if and only if all columns of τ indexed by x have value a, *i.e.* τ must be coherent for x and those columns must have value a.

Lemma 2 [Selection Wellformed] *For all well-formed relations $\langle Y, R_Y \rangle$ and all x, and all $a \in \mathbb{Z}$, the relation $\langle Y, select_{x=a}R_y \rangle$ is well-formed.*

Finally, we define *coherent selection* as follows.

$$select_Y(R_X) \overset{def}{=} \{\tau \in R_X | \mathbf{coh}\{X, Y\}(\tau)\}$$

Coherent selection selects the tuples which are coherent with respect to Y.

2.5.3 Projection

Projection is an operation for creating new relations from existing ones by allowing for the deletion, reordering and duplication of columns. We use a generalized version here that allows duplicate names. This is because many constraint solvers (including Minion [14]) allow schemata to contain duplicate names.

Lemma 3 [Projection maps exist] *For all vectors X and Y, if $Y \subseteq X$, then there exists a function from the indexes of Y to the indexes of X (say $f \in \{0..|Y|-1\} \to \{0..|X|-1\}$) such that*

$$\forall i : \{0..|Y|-1\}.\ Y[i] = X[f(i)]$$

Note that there is no restriction on the relative lengths of X and Y, *e.g.* it is possible for any of the following to hold: $|Y| < |X|$, $|Y| = |X|$ or $|Y| > |X|$. The projection maps are evidence witnessing claims of the form $Y \subseteq X$. Furthermore, because our model allows for duplicated columns, there may be multiple projection maps witnessing an inclusion $Y \subseteq X$.

Example 2 Consider

$$Y = \langle x_4, x_2, x_2, x_1, x_3\rangle \; X = \langle x_1, x_2, x_3, x_4\rangle$$

then $Y \subseteq X$ is witnessed by the projection map:

$$\{\langle 0,3\rangle, \langle 1,1\rangle, \langle 2,1\rangle, \langle 3,0\rangle, \langle 4,2\rangle\}$$

Similarly, $X \subseteq Y$ and is witnessed by the following.

$$\{\langle 0,3\rangle, \langle 1,1\rangle, \langle 2,4\rangle, \langle 3,0\rangle\}$$

Also $\langle x_2\rangle \subseteq Y$ is witnessed by two functions.

$$\{\langle 0,1\rangle\} \; \{\langle 0,2\rangle\}$$

Lemma 4 [Tuple Projection] *Given X and Y, if $Y \subseteq X$ is witnessed by f, for each X-tuple τ there is a vector $Y_f(\tau) : \{0..|Y|-1\} \to \mathbb{Z}$ such that*

$$\forall i : \{0..|Y|-1\}. Y_f(\tau)[i] = \tau[f(i)]$$

Corollary 1 [Tuple Projection Wellformed] *Given X and Y, if $Y \subseteq X$ is witnessed by f, for each X-tuple τ, $Y_f(\tau)$ is a Y-tuple ,* i.e. $|Y_f(\tau)| = |Y|$ *and all values in $Y_f(\tau)$ are in their domains.*

Whenever $Y \subseteq X$, projection maps f and g witnessing this fact behave the same when used to index into tuples coherent with Y. This is illustrated by the following example.

Example 3 Suppose $Y = \langle x, y\rangle$ and $X = \langle x, x, w, y, w\rangle$ then there are two projections maps witnessing $Y \subseteq X$, $f = \{\langle 0,0\rangle, \langle 1,3\rangle\}$ and $g = \{\langle 0,1\rangle, \langle 1,3\rangle\}$. Now, any length $|X| = 5$ tuple coherent with Y is of the form $\tau = \langle a, a, b, c, d\rangle$ where $a, b, c, d \in \mathbb{Z}$. Thus, even though $f(0) \neq g(0)$ the following equalities hold:

$$\tau[f(0)] = \tau[0] = a = \tau[1] = \tau[g(0)]$$

This observation is made precise by the following lemma.

Lemma 5 [Coherent Projection Unique] *For all X and Y, and for all projection maps f and g witnessing $Y \subseteq X$, for all X-tuples τ coherent with schema Y, $Y_f(\tau) = Y_g(\tau)$.*

Notational Remark 1 *Since projections Z where $Z \subseteq X$ do not depend on the projection map they are built from when the X-tuple τ is coherent with Z, we will simply write $Z(\tau)$ in this case.*

Lemma 6 [Projection Coherent] *For all X, Y and Z, if $Y \subseteq X$ and if τ is an X-tuple coherent with Z, then $Y[\tau]$ is a Y-tuple coherent with Z.*

We lift the notation tuple-wise to relations as given by the following definition.

Definition 4 [Relation Projection] Given X and Y, and a wellformed relation $\langle X, R_X\rangle$, if $Y \subseteq X$ is witnessed by f,

$$Y_f(\langle X, R_X\rangle) = \langle Y, \{\tau \in \mathbb{Z}^{|Y|} \mid \exists \tau' \in R_X.\; \tau = Y_f(\tau')\}\rangle$$

Lemma 7 [Relation Projection WF] *For all well-formed relations $\langle X, R_X\rangle$ and all Y, $Y \subseteq X$ having a witnesses f, the relation $Y_f\langle Y, R_Y\rangle$ is well-formed.*

Now we are able to define an equivalence of constraints which does not depend on the ordering of schemata.

Definition 5 [Schema Equivalence]

$$X \equiv Y \overset{def}{=} X \subseteq Y \wedge Y \subseteq X$$

Definition 6 [Constraint Equivalence]

$$\begin{aligned}
&\langle X, R_X\rangle \equiv \langle Y, R_Y\rangle \overset{def}{=}\\
&\quad X \equiv Y \wedge\\
&\quad X \subseteq Y \text{ is witnessed by the projection map } f \wedge\\
&\quad \langle Z, R_Z\rangle = Y_f(\langle X, R_X\rangle)\wedge\\
&\quad \mathit{select}_Z(\langle Z, R_Z\rangle) = \mathit{select}_Y(\langle Y, R_Y\rangle)
\end{aligned}$$

There are three steps to the constraint equivalence definition. First, it is required that the schemata are equivalent. Then a projection is used to reorder the schema X to match Y, creating the new relation $\langle Z, R_Z\rangle$. Finally, coherent selection is used to remove incoherent tuples. The two constraints are equivalent if they have the same set of coherent tuples.

Constraints are rarely presented extensionally but are instead described in some syntactic way. We introduce the following notation to denote the map from syntactic descriptions to their extensional meanings.

Definition 7 (Semantics) Given a syntactic description of a constraint (say $\mathcal{C}$) over schema X and where σ is a signature consistent with X, we will write $[\![\mathcal{C}]\!]_\sigma$ to denote its extension.

3 Propagation and Support

Propagation is the process of narrowing the domains of variables so that solutions are preserved. This effectively shrinks the search-space and is one of the fundamental techniques used in constraint programming. It has been described ([13, pp. 17]) as a process of *inference* to distinguish it from *search.*

Definition 8 [Generalized Arc Consistency] Given a constraint $\mathcal{C}$ with schema X and a signature σ, we say $\sigma' \sqsubseteq \sigma$ is *Generalized Arc Consistent* iff

$$\forall i \in \{0..|X|-1\}.\ \forall k \in \sigma'(X[i]).\ \exists \tau \in [\![\mathcal{C}]\!]_\sigma.\ \tau[i] = k$$

If σ is Generalized Arc Consistent, we say it is *GAC.*

Corollary 2 [Generalized Arc Consistency] *Given a constraint $\mathcal{C}$ and a signature σ, σ is GAC for $\mathcal{C}$ iff*

$$\forall \sigma' \sqsubset \sigma.\ [\![\mathcal{C}]\!]_{\sigma'} \subset [\![\mathcal{C}]\!]_\sigma$$

i.e. *if all signatures having strictly narrower domains provide strictly fewer solutions for $\mathcal{C}$ than σ.*

Typically, an individual constraint may be implemented by a collection of propagators.

3.1 Support

The concept of support was introduced in Section 1.4. Support is *evidence* that a set of domain values (or a single value) are consistent for some definition of consistency (for example, GAC) for a particular constraint C. If a set of values have no support, then they cannot be part of any solution to C, and therefore can be eliminated from variable domains without losing any solutions to the CSP. The concept of support is central to the process of propagation.

In [5, pp. 37] Bessière gives a description of when a tuple supports a literal. We use a more expressive model where support (or perhaps we should call it *evidence*) is defined by sets of tuples. In most cases, supports will be singletons (*i.e.* they are simply represented by a set containing a single tuple). However, some constraints require a set of tuples to express the condition for support.

Example 4 Consider the constraint $\mathcal{C} = \text{AllDifferent}(x_1, x_2, x_3)$ with the signature $\sigma : x_1 \in \{1,2\}$, $x_2 \in \{1,2,3,4\}$, $x_3 \in \{1,2,3,4,5\}$. This signature is GAC. Given Bessière's description of support [5, pp. 37] (as used by general-purpose GAC algorithms such as GAC-Schema [7]), each literal in the signature would be supported by a tuple containing the literal. Hence every literal is contained in the support for $\mathcal{C}$. However, not all literals are required; the following set is sufficient: $L = \{\langle x_1, 1\rangle, \langle x_1, 2\rangle, \langle x_2, 2\rangle, \langle x_2, 4\rangle, \langle x_3, 2\rangle, \langle x_3, 3\rangle, \langle x_3, 5\rangle\}$ [16, §5.2]. While all literals in L remain valid, in some smaller signature $\sigma_1 \sqsubseteq \sigma$, then the constraint remains GAC. This can be used to avoid calling the propagator, and therefore is important to capture in our definition of generalized support.

Extensional constraints (sets of tuples) are interpreted disjunctively, *i.e.* as long as the set is non-empty, a solution exists. Similarly, support exists if the support set is non-empty. Our generalization of support is to model it as a set of tuples interpreted conjunctively *i.e.* thay all must be valid for support to exist. Thus, a generalized support set is a disjunction of conjunctions ($\exists\forall$); we say support exists if at least one support is present in the set and all the tuples in that support are valid w.r.t. variable domains.

We use the following as a simple running example throughout this section.

Example 5 Consider the constraint $x + y + z \geq 2$ with initial signature $\sigma : x, y, z \in \{0,1\}$. The signature is GAC, and the constraint is satisfied by three tuples: $[\![x + y + z \geq 2]\!]_\sigma = \{\langle 0,1,1\rangle, \langle 1,0,1\rangle, \langle 1,1,0\rangle\}$.

3.1.1 Support Sets

Definition 9 [Support property] Given a schema Y and signature σ over Y, a *support property* is a predicate

$$P : \textit{signature} \rightarrow 2^{\mathbb{Z}^{|Y|}} \rightarrow \mathbb{B}$$

mapping signatures and sets of integer tuples of length $|Y|$ to a Boolean. We will sometimes write the parameter indicating which signature $P[\sigma]$ depends on as a subscript P_σ or drop it entirely if the property does not depend on a signature.

Definition 10 **[Support Set for a property** P**]** Given a schema Y and a signature σ over Y and a property of sets of Y-tuples, P_σ we define the support set for P to be the set:

$$\begin{aligned} &\mathit{support}_{\langle Y,\sigma\rangle}(P) \overset{def}{=} \\ &\qquad \{S \subseteq Y\text{--}\mathit{tuple}_\sigma | P_\sigma(S) \wedge \forall S' \subset S.\ \neg P_\sigma(S')\} \end{aligned}$$

Note that support sets are minimal w.r.t. the property P since they contain no subset which also satisfies the property.

Consider example 5, the constraint $x + y + z \geq 2$. One support property is $P_\sigma(S) \overset{def}{=} \exists\tau \in S : \sum\tau \geq 2 \wedge \tau[0] = \min(\sigma(x))$. This property admits sets of tuples of any size as long as one tuple satisfies the constraint, and the value for x in that tuple is the minimum value in $\sigma(x)$[2]. The support set for P_σ is $\mathit{support}_{\langle\langle x,y,z\rangle,\sigma\rangle}(P) = \{\{\langle 0,1,1\rangle\}\}$.

A collection of properties is supported if they all are.

Definition 11 **[Support for a collection of properties]** If $\mathcal{P} = \{P_1, \cdots, P_k\}$ is a collection of properties sharing schema Y and σ is a signature over Y, we write

$$\mathit{support}_{\langle Y,\sigma\rangle}(\mathcal{P}) \overset{def}{=} \forall P \in \mathcal{P}.\ \mathit{support}_{\langle Y,\sigma\rangle}(P) \neq \emptyset$$

3.1.2 Admissible Properties and Triggers

Our language for properties is unrestrained and allows us to specify properties that are not sensible for specifying propagators. Therefore an admissibility condition is required. We define *p-admissibility* as follows.

Definition 12 **[P-Admissibility]** We say a property P is *p-admissible* if it satisfies the following condition.

$$\begin{aligned} &\forall\sigma.\ \forall\sigma' \sqsubseteq \sigma. \\ &\quad \forall S \subseteq Y\text{--}\mathit{tuple}_\sigma. \\ &\qquad (P_\sigma(S)\ \wedge\ S \subseteq Y\text{--}\mathit{tuple}_{\sigma'}) \Rightarrow P_{\sigma'}(S) \end{aligned}$$

In this case, we write $p\text{--}\mathit{admissible}(P)$.

P-admissibility is a kind of stability condition on properties that guarantees that if a $P_\sigma(S)$ holds and the domain is narrowed to σ', but no tuple is lost from S because of the narrowing, then $P_{\sigma'}(S)$ must also hold. In the implementation of dynamic-triggered propagators [15], it is implicitly assumed that support for these propagators satisfy this property.

Continuing example 5, the support property $P_\sigma(S) \overset{def}{=} \exists\tau \in S : \sum\tau \geq 2 \wedge \tau[0] = \min(\sigma(x))$ is p-admissible: $\sum\tau \geq 2$ does not depend on σ, and $\tau[0] = \min(\sigma(x))$ can only be falsified under σ' when the value $\min(\sigma(x))$ is not in $\sigma'(x)$. This means τ is not in $\langle x,y,z\rangle\text{--}\mathit{tuple}_{\sigma'}$ so the implication is trivially satisfied. Suppose $S = \{\langle 0,1,1\rangle\}$. The only way $P_{\sigma'}(S)$ can be false is if $0 \notin \sigma'(x)$. In this

[2] This support property corresponds to a propagator that prunes the minimum value of x whenever there is no supporting tuple containing it. To enforce GAC, two other properties would be required for y and z.

case, S contains a tuple that is not valid in σ' therefore the p-admissibility property is trivially true.

A constraint solver has a trigger mechanism which calls propagators when necessary. Each propagator registers an interest in domain events by *placing triggers*. For example, if a propagator placed a trigger on $\langle x, a\rangle$, then the removal of value a in $\sigma(x)$ would cause the propagator to be called. (This is named a *literal trigger* [15], or *neq event* [24].)

In this paper, we focus on literal triggers which can be moved during search. We consider two different types of movable literal trigger: those which are restored as search backtracks (named *dynamic literal triggers*), and those which are not restored (named *watched literals* [15]).

The definition of p-admissibility allows the use of dynamic literal triggers, among other types. Watched literals are preferable to dynamic literal triggers because there is no need to restore them when backtracking, which saves space and time. However, it is not always possible to apply watched literals. We define an additional condition on properties named *backtrack stability*, which is sufficient to allow the use of watched literals.

Definition 13 **[Backtrack Stability]** We say a property P is *backtrack stable* if it satisfies the following condition.

$$\begin{array}{l}\forall S.\, \forall \sigma.\, \forall \sigma' \sqsubseteq \sigma. \\ \quad S \neq \emptyset \Rightarrow \\ \qquad P_{\sigma'}(S) \Rightarrow P_{\sigma}(S)\end{array}$$

Backtrack stability states that any non-empty support S under σ' must remain a support for all signatures σ where σ is larger than σ'. This guarantees that a non-empty support S will remain valid as the search backtracks. The empty support indicates that the property is trivially satisfied; this support is not usually valid after backtracking, so it is excluded here.

Continuing example 5, the support property $P_\sigma(S) \overset{def}{=} \exists \tau \in S : \sum \tau \geq 2 \wedge \tau[0] = \min(\sigma(x))$ is not backtrack stable because $\min(\sigma(x))$ may not be the same as $\min(\sigma'(x))$.

Backtrack stability is in fact too strong: it is not necessary for a support to remain valid for *all* larger signatures, it is only necessary for it to remain valid at signatures that are reachable on backtracking. However it is sufficient for the purposes of this paper.

Backtrack stability also depends on the form of properties. The element support properties presented in Section 4.1.1 are not backtrack stable. However, they can be reformulated to be backtrack stable, by dividing them up as we show in Section 4.1.2.

For some property $P_\sigma(S)$ the support S is *evidence* that the constraint corresponding to P is consistent. The intuition is that S remains valid evidence until domains are narrowed to the extent that $S \not\subseteq Y\text{–}tuple_{\sigma'}$ (where $\sigma' \sqsubseteq \sigma$). This is an efficiency measure: a constraint solver can disregard the constraint corresponding to P until $S \not\subseteq Y\text{–}tuple_{\sigma'}$.

For example, the property $P_\sigma(S) \overset{def}{=} \forall b \notin \sigma(j).\langle i, b\rangle \in S$ is not p-admissible when $j \neq i$.

Definition 14 **[Properties *True* and *False*]** We define the constant properties *True* and *False* by lifting them to functions of sets of tuples.

$$True(S) = True \; False(S) = False$$

Lemma 8 **[True singleton]** *For all Y and for every signature σ over Y,*

$$support_{\langle Y,\sigma\rangle}(True) = \{\emptyset\}$$

Note, that it might be assumed that if any of the domains in σ are empty, then there should be no support, even for the *True* property. Checking for emptiness is not a function of support, but is done at a higher level.

Lemma 9 **[False Empty]** *For all Y and for every signature σ over Y,*

$$support_{\langle Y,\sigma\rangle}(False) = \emptyset$$

Corollary 3 **[*True* and *False* are p-Admissible]** *The properties True and False are p-Admissible.*

We can combine supports by taking the conjunctions or disjunctions of their properties.

Definition 15 We define the conjunction and disjunction of support properties as follows.

$$\begin{array}{rcl}(P \wedge Q)_\sigma(S) & \overset{def}{=} & P_\sigma(S) \wedge Q_\sigma(S) \\ (P \vee Q)_\sigma(S) & \overset{def}{=} & P_\sigma(S) \vee Q_\sigma(S)\end{array}$$

We state the following lemma without proof.

Lemma 10 **[Conjunction and disjunction p-admissible]** *Given a schema Y and signature σ for Y and two p-admissible properties P and Q, then $(P \wedge Q)$ and $(P \vee Q)$ are p-admissible as well.*

3.1.3 Extensional Support for literals

Definition 16 **[Support Property (for a Literal)]** Given a relation schema Y and signature σ over Y, and a literal $\langle i = a\rangle$, then: $\langle i = a\rangle$ denotes the property supporting this literal and is given by:

$$\langle i = a\rangle(S) \overset{def}{=} \exists\tau \in S.\, \tau[i] = a$$

The support for $\langle i = a\rangle$ is just the set $support_{\langle Y,\sigma\rangle}(\langle i = a\rangle)$.

Corollary 4 *If $S \in support_{\langle Y,\sigma\rangle}(\langle i = a\rangle)$ then S is a singleton.*

Proof Assume $S \in support_{\langle Y,\sigma\rangle}(\langle i = a\rangle)$ then $\langle i = a\rangle(S)$ holds, *i.e.* we know $\exists\tau \in S.\tau[i] = a$. Thus $|S| \geq 1$. Now, we assume that $|S| > 1$ and show a contradiction. There is at least one tuple in S, such that $\tau[i] = a$. If there is any other tuple $\tau' \in S$ where $\tau \neq \tau'$ then $\langle i = a\rangle(S - \{\tau'\})$ holds as well, and since this set is smaller, S was not minimal and so was not a support as we assumed.

Lemma 11 **[Literals are p-admissible]** *Given a schema Y and a signature σ on Y, if $i \in \{0..|Y| - 1\}$ and $a \in \sigma(Y[i])$ then $\langle i = a\rangle$ is a P-admissible property.*

Proof Note that $\langle i = a\rangle$ does not refer to σ at all and so is P-admissible.

3.1.4 Structural Support - Evidence

Literal support captures support for variable-value pairs. Structural support is support for some structural condition not representable by a single tuple. Thus, if any tuple in a structural support is lost, then the support no longer holds. In example 4 (GAC AllDifferent) we gave a list of literals as evidence that an AllDifferent constraint is GAC. Using the support property for a literal, these would be represented as a structural support in our framework.

3.2 Soundness and Completeness of a Collection of Propagators

Propagators narrow domains to minimize the search space and provide evidence that the narrowed domains have not eliminated any solutions. Constraints may be supported by a collection of propagators. To show that the propagators are correct with respect to the constraint they support we show they are sound and complete.

3.2.1 Soundness

Definition 17 [Propagator Soundness] Given a constraint C with schema Y and a set of propagators $\mathcal{P} = \{P_1, \cdots, P_m\}$ we say $\mathcal{P}$ is *sound* with respect to the constraint C if the following holds:

$$\forall \sigma.\ \text{singleton}(\sigma) \Rightarrow (\mathit{support}_{\langle Y,\sigma\rangle}(\mathcal{P}) \Rightarrow [\![C]\!]_\sigma \neq \emptyset)$$

Soundness says that for the most restricted non-empty signatures (ones where all domains in the signature have been narrowed to a singleton) the propagator must be able to distinguish between the constraint being empty or inhabited by a single tuple. If support is non-empty at a singleton domain then the constraint must be true there as well. The definition of soundness presented here is related to the one in [25].

Thinking of support as evidence for truth, one might expect soundness to be characterized as follows:

$$\forall \sigma.\ \mathit{support}_{\langle Y,\sigma\rangle}(\mathcal{P}) \Rightarrow [\![C]\!]_\sigma \neq \emptyset$$

This is too strong. At a non-singleton signature, support is an approximation to truth. For example, even though a constraint may fail in a particular non-convex domain (*i.e.* the domain has gaps), a propagator that operates on domain bounds may not recognize the domain is not convex until the signature has been narrowed further.

3.2.2 Completeness

Completeness guarantees that if the meaning of a constraint is non-empty at a signature σ (semantic truth) then there is support for the family of properties $\mathcal{P}$. The wrinkle on this scheme is that the support may not exist at σ itself, but only at some refined $\sigma' \sqsubseteq \sigma$. If so, we insist that the constraint has not lost any tuples at the refined signature σ'.

Definition 18 [Propagator Completeness] Given a constraint C with schema Y and a set of propagators $\mathcal{P} = \{P_1, \cdots, P_m\}$ we say $\mathcal{P}$ is *complete* with respect to the constraint C if the following holds:

$$\begin{array}{l} \forall\sigma.\ [\![C]\!]_\sigma \neq \emptyset \ \Rightarrow \\ \quad \exists\sigma' \sqsubseteq \sigma.\quad [\![C]\!]_\sigma \subseteq [\![C]\!]_{\sigma'} \ \wedge\ \mathit{support}_{\langle Y,\sigma'\rangle}(\mathcal{P}) \end{array}$$

If $\mathcal{P}$ is complete we write *complete*$(\mathcal{P})$.

Theorem 1 [Local Completeness] *Give a set of properties $\mathcal{P} = \{P_1, \cdots, P_k\}$ defined over schema Y, if each singleton $\{P_i\}$ is complete then $\mathcal{P}$ is complete.*

Proof If $\mathcal{P}$ is supported at σ, then use witness σ for σ' and completness trivially holds. Suppose there is not support for $\mathcal{P}$ at σ where $[\![C]\!]_\sigma \neq \emptyset$. Choose one of the $P_i \in \mathcal{P}$ such that $\neg\mathit{support}_{\langle Y,\sigma\rangle}(P_i)$ and let $\sigma', \sigma' \sqsubset \sigma$ be the signature claimed to exist in the proof of completeness of P_i. By completeness of $\{P_i\}$, $[\![C]\!]_\sigma \subseteq [\![C]\!]_{\sigma'}$. If there is support for $\mathcal{P}$ at σ' then $\mathcal{P}$ is complete. If not, iterate this process by choosing another $P_k \in \mathcal{P}$ that is not supported at σ'. The fixed-point of this process must yield a signature $\hat{\sigma}$ such that $\mathit{support}_{\langle X,\hat{\sigma}\rangle}(\mathcal{P})$. The fixed-point exists because $\sqsubset$ is a well-founded relation on signatures.

Our definition of completeness ensures that a propagator derived from a support property does not fail early, therefore it is merely a correctness property. It is similar in intention to Maher's definition of weak completeness [22], although Maher's definition only applies to singleton domains.

Soundness and completeness as defined here are not the only options for characterizing the correctness of a set of generalized support properties. In [15] it is shown that properties imply the domain is GAC (def. 8). Other forms of consistency could also serve as correctness conditions for a set of properties.

3.3 Formal Development of Constraint Propagators

The methodology for formal development of propagators for a constraint C is as follows:

i. Describe support properties ($\mathcal{P} = \{P_1, \cdots, P_k\}$) that characterize constraint C and prove that they are p-admissible.
ii. For each property P_i, give a constructive proof of the propagation schema given in Def. 19. The computational content of these proofs gives correct-by-construction algorithms for each propagator.
iii. Prove the soundness and completeness of $\mathcal{P}$ with respect to C. This shows the collection of propagators are correct w.r.t. the constraint C. This proof often reuses the propagation schema proofs.

3.3.1 The Propagation Schema

We present the following schematic formula whose constructive proofs capture the methods of generating support for a particular property P.

Definition 19 [Propagation Schema] Given a schema X, a signature σ and a p-admissible property P, constructive proofs of the following statement yield a propagator for P.

$$\begin{array}{l}
\forall S \in \mathit{support}_{\langle X,\sigma\rangle}(P).\\
\quad \forall \sigma_1 \sqsubseteq \sigma.\, \text{nonempty}(\sigma_1) \Rightarrow\\
\quad\quad S \notin \mathit{support}_{\langle X,\sigma_1\rangle}(P) \Rightarrow\\
\quad\quad\quad (\exists \sigma_2 \sqsubseteq \sigma_1.\, \text{nonempty}(\sigma_2) \wedge\\
\quad\quad\quad\quad \exists S' \in \mathit{support}_{\langle X,\sigma_2\rangle}(P).\\
\quad\quad\quad\quad\quad \forall \sigma_3.\, \sigma_2 \sqsubset \sigma_3 \sqsubseteq \sigma_1 \Rightarrow \mathit{support}_{\langle X,\sigma_3\rangle}(P) = \emptyset)\\
\quad\quad\quad \vee \big(\forall \sigma_2 \sqsubseteq \sigma_1.\, \text{nonempty}(\sigma_2) \Rightarrow \mathit{support}_{\langle X,\sigma_2\rangle}(P) = \emptyset\big)
\end{array}$$

We are interested in constructive proofs[3] of the propagator schema when P is instantatied to individual support properties.

Given an admissible support property P, a constructive proof of the propagator schema yields a function that takes as input a set S, evidence that $S \in \mathit{support}_{\langle X,\sigma\rangle}(X)$, a signature σ_1 and evidence that $\sigma_1 \sqsubseteq \sigma$, evidence that $S \notin \mathit{support}_{\langle X,\sigma_1\rangle}(P)$ and returns one of two items:

i.) a new signature σ_2, together with evidence that $\sigma_2 \sqsubseteq \sigma_1$, a set of tuples S' and evidence that $S' \in \mathit{support}_{\langle X,\sigma_2\rangle}(P)$ and evidence that σ_2 is maximal.
ii.) Evidence that there is no support for P in σ_1 or for any smaller signature.

Lemma 12 [non-empty in propagation schema] *In the propagation schema, if we assume the antecedent $S \notin \mathit{support}_{\langle X,\sigma_1\rangle}(P)$ for $\sigma_1 \sqsubseteq \sigma$ then $S \neq \emptyset$.*

Proof By p-admissibility of P, if $\emptyset \in \mathit{support}_{\langle X,\sigma\rangle}(P)$ then for all $\sigma_1 \sqsubseteq \sigma$, $\emptyset \in \mathit{support}_{\langle X,\sigma_1\rangle}(P)$.

4 Generating Propagators

In this section we present two case studies of applying our methodology.

4.1 A Propagator for the Element Constraint

The element constraint is widely useful in specifying a large class of constraint problems. It has the form `element`(X, y, z) where X is a vector of variables and y and z are variables. The meaning of the element constraint is the set of all coherent tuples on the schema $\langle X \cdot y \cdot z\rangle$ of the following form.

$$\tau = \langle v_1, \cdots, v_{i-1}, j, v_{i+1}, \cdots, v_k, i, j\rangle$$

Thus, $\tau[k + 1] = i$ indexes $\langle v_1, \cdots, v_k\rangle$ and $\tau[k + 2] = \tau[i]$.

[3] There is a classical proof of propagator schema that is independent of the property P and carries no interesting computational content.

Definition 20 [Element Semantics]

$$
\begin{aligned}
&[\![\texttt{element}(X, y, z)]\!]_\sigma = \langle\langle X \cdot y \cdot z\rangle, R\rangle \\
&\quad \text{where} \\
&\quad R = \{\tau \in \langle X \cdot y \cdot z\rangle\text{–}tuple_\sigma \mid \\
&\qquad k = |X| \wedge \tau[k+1] \in \{1..k\} \wedge \tau[k+2] = \tau[\tau[k+1]]\}
\end{aligned}
$$

The element constraint is widely used because it represents the very basic operation of indexing a vector [18]. For example, Gent et al. model Langford's number problem and quasigroup table generation problems using element [15].

In [15, pp. 188] three properties to establish GAC for the element constraint are characterized. We restate theorem 1 from that paper here:

Theorem 2 [Theorem 1 of reference [15]**.]** *Given an element constraint of the form* $\texttt{Element}(X, y, z)$*, domains given by a signature* σ *are Generalized Arc Consistent if and only if all of the following hold.*

$$\forall i \in \sigma(y).\ \sigma(y) = \{i\} \Rightarrow \sigma(X[i]) \subseteq \sigma(z) \tag{1}$$
$$\forall i \in \sigma(y).\ \sigma(X[i]) \cap \sigma(z) \neq \emptyset \tag{2}$$
$$\sigma(z) \subseteq \bigcup_{i \in \sigma(y)} \sigma(X[i]) \tag{3}$$

4.1.1 Support Properties

Each of the three properties above can be characterized as properties of their generalized supports.

Definition 21 [Element Support Properties] Given a schema X and variables y and z and a signature σ there are three properties corresponding to three propagators for establishing GAC for the element constraint $\texttt{Element}(X, y, z)$. Let k be $|X|$, then $k+1$ is the index of y and $k+2$ is the index of z in the schema $(X \cdot y \cdot z)$.

$$
\begin{aligned}
&P_1[\sigma](S) \overset{def}{=} \\
&\quad (\exists i, j : \sigma(y).\quad i \neq j\ \wedge\ \langle k+1, i\rangle \in S \wedge \langle k+1, j\rangle \in S) \\
&\quad \vee\ \ \forall i : \sigma(y).\ \forall a : \sigma(X[i]).\ \langle k+2, a\rangle \in S \\
&P_2[\sigma](S) \overset{def}{=} \forall i : \sigma(y).\ \exists a : \sigma(z).\ \langle i, a\rangle \in S \wedge \langle k+2, a\rangle \in S \\
&P_3[\sigma](S) \overset{def}{=} \forall a : \sigma(z).\ \exists i : \sigma(y).\ \langle i, a\rangle \in S \wedge \langle k+1, i\rangle \in S
\end{aligned}
$$

Note that for property P_1, the first disjunct is true iff the domain of the index variable y has more than one element, $|\sigma(y)| > 1$. Support for this disjunct is a pair of literals $\langle k+1, i\rangle$ and $\langle k+1, j\rangle$ where $i, j \in \sigma(y),\ i \neq j$ [4]. Logically, $(\exists i, j : \sigma(y).\quad i \neq j)$ is equivalent, but for our purposes we must provide p-admissible support. Once the domain of the index variable is a singleton ($\sigma(y) = \{i\}$), the second disjunct of P_1 may be satisfied. This disjunct is supported by a set of $|\sigma(X[i])|$ literals of the form $\langle k+2, a\rangle$, one literal for each $a \in \sigma(X[i])$. This is evidence for $\sigma(X[i]) \subseteq \sigma(z)$ since $k+2$ is the index of z in the schema $(X \cdot y \cdot z)$.

[4] This specification corresponds to a set of dynamic literal triggers [15]. Ideally a static assignment trigger would be used for P_1, which would trigger the propagator when y is assigned. However, assignment triggers are outside the scope of this paper.

Property P_2 is supported iff $\sigma(X[i]) \cap \sigma(z)$ is non-empty for every $i \in \sigma(y)$. The support is $2m$ literals where $m = |\sigma(y)|$, two for each $i \in \sigma(y)$. These have the form $\langle i, a\rangle$ and $\langle k+2, a\rangle$ where a is some value in $\sigma(z)$. If there is no support, then $\sigma(X[i]) \cap \sigma(z) = \emptyset$.

Property P_3 is supported iff $\sigma(z) \subseteq \bigcup_{i\in\sigma(y)} \sigma(X[i])$. The support is a set of $2m$ literals where $m = |\sigma(z)|$, two for each $a \in \sigma(z)$. The literals have the form $\langle i, a\rangle$ and $\langle k+1, i\rangle$ where i is some value in $\sigma(y)$. If there is no support then for some $a \in \sigma(z)$, for all i $a \notin \sigma(X[i])$.

It is easy to prove that the three properties act as intended:

Theorem 3 *Given a signature σ, we have:*

- *(1) is true if and only if $\exists S : P_1[\sigma](S)$*
- *(2) is true if and only if $\exists S : P_2[\sigma](S)$*
- *(3) is true if and only if $\exists S : P_3[\sigma](S)$*

Proof The if directions are all easy. For (1), if the first disjunct of P_1 is satisfied then $|\sigma(y)| > 1$ so (1) is vacuous. If the second disjunct is satisfied, it ensures that $\sigma(X[i]) \subseteq \sigma(z)$. If $P_2(S)$ is true then, for each element of the domain of the index variable y, there is a value $a \in \sigma(X[i]) \cap \sigma(z)$, establishing (2). If $P_3(S)$ is true then, for any value a in $\sigma(z)$ there is a value i of the index variable with $a \in \sigma(X[i])$, proving that (3) holds.

For Only if, first suppose that (1) is true. If $|\sigma(y)| > 1$ then we can find i, j to satisfy the first disjunct of P_1, and set $S = \{\langle k+1, i\rangle, \langle k+1, j\rangle\}$. Otherwise, we have $\sigma(y) = \{i\}$ and $\sigma(X[i]) \subseteq \sigma(z)$. We can thus set $S = \{\langle k+2, a\rangle | a \in \sigma(X[i])\}$.

Suppose (2) is true. We have $\sigma(X[i]) \cap \sigma(z) \neq \emptyset$ for each $i \in \sigma(y)$. So for each i there is thus some a_i with $a_i \in \sigma(X[i]) \cap \sigma(z)$. We can thus set $S = \{\langle i, a_i\rangle, \langle k+2, a_i\rangle | i \in \sigma(y)\}$.

Suppose (3) is true. Since $\sigma(z) \subseteq \displaystyle\bigcup_{i\in\sigma(y)} \sigma(X[i])$, we have for each $a \in \sigma(z)$ some i_a such that $i_a \in \sigma(y)$ and $a \in \sigma(X[i_a])$. We can thus set $S = \{\langle i_a, a\rangle, \langle k+1, i_a\rangle | a \in \sigma(z)\}$.

4.1.2 P-Admissibility and Backtrack Stability

Following our methodology, we first prove that properties P_1, P_2 and P_3 are p-admissible.

Lemma 13 [**P_1 is p-admissible**]

$$P\!-\!admissible(P_1)$$

Proof We case split on the disjuncts of P_1. The first disjunct requires distinct values $i, j \in \sigma(y)$. Assuming $S \subseteq \langle X \cdot y \cdot z\rangle\!-\!tuple_{\sigma'}$, $i, j \in \sigma'(y)$ because the two necessary literals are in S, therefore $P_1[\sigma'](S)$ holds.

For the second disjunct of P_1, since $\sigma' \sqsubseteq \sigma$ we can see that $\sigma'(y) \subseteq \sigma(y)$ and $\forall i. \sigma'(X[i]) \subseteq \sigma(X[i])$, therefore all necessary literals are present in S and $P_1[\sigma'](S)$ holds.

Lemma 14 [**P_2 is p-admissible**]

$$P\!-\!admissible(P_2)$$

Proof Since $\sigma' \sqsubseteq \sigma$, $\sigma'(y) \subseteq \sigma(y)$ therefore there are fewer (or the same) values of i to consider under σ'. Assuming $S \subseteq \langle X \cdot y \cdot z\rangle\textit{–tuple}_{\sigma'}$, for each i, $\langle k+2, a\rangle \in S$ therefore $a \in \sigma'(z)$ and $P_2[\sigma'](S)$ holds.

Lemma 15 [**P_3 is p-admissible**]

$$P\textit{–admissible}(P_3)$$

Proof The proof is the same as above, with z and y exchanged, i and a exchanged, and $k+1$ substituted for $k+2$.

P_1, P_2 and P_3 are not backtrack stable according to Def. 13. However, P_2 and P_3 can be straightforwardly reformulated to be backtrack stable: the universal quantifier is expanded to a conjunction using the initial signature, then each conjunct is made into an individual property, subscripted by i or a respectively. For example, P_2 is transformed as follows.

$$\begin{aligned} P_{2,i}[\sigma](S) \overset{def}{=} i \in \sigma(y) \Rightarrow (&\exists a : \sigma(z).\ \langle i, a\rangle \in S \\ &\wedge \langle k+2, a\rangle \in S) \end{aligned}$$

Each of these smaller properties then requires two literals as support, or (if $i \notin \sigma(y)$) the empty set, and they are backtrack stable. P_1 can be reformulated to be backtrack stable, by expanding out the universal quantifiers in the same way as for P_2. P_1 would be subscripted by i and a, $\forall i : \sigma(y)$ replaced with $i \in \sigma(y) \Rightarrow$, and the same for $\forall a : \sigma(X[i])$. These reformulations give a large set of properties, so for the sake of simplicity we use the original P_1, P_2 and P_3.

4.1.3 Proofs of the Propagation Schema

Now that we have established p-admissibility for each of P_1, P_2 and P_3 we prove the instances of the propagator schema for each of them.

Theorem 4 (**P_1 Support Generation**) *We consider P_1 on constraint* `Element`(X, y, z). *We claim that Def. 19 (propagation schema) holds for P_1.*

Proof Let C be an element constraint of the form `Element`(X, y, z) where $|X| = k$ and let σ and σ_1 be signatures mapping the variables in $X.y.z$ to their respective domains. We claim the following:

$$\begin{aligned}
&\forall S \in \textit{support}_{\langle X,\sigma\rangle}(P_1). \\
&\ \ \forall \sigma_1 \sqsubseteq \sigma. \\
&\ \ \ \ \text{nonempty}(\sigma_1) \Rightarrow \\
&\ \ \ \ \ \ S \notin \textit{support}_{\langle X,\sigma_1\rangle}(P_1) \Rightarrow \\
&\ \ \ \ \ \ \ \ (\exists \sigma_2 \sqsubseteq \sigma_1.\, \text{nonempty}(\sigma_2) \wedge \\
&\ \ \ \ \ \ \ \ \ \ \exists S' \in \textit{support}_{\langle X,\sigma_2\rangle}(P_1). \\
&\ \ \ \ \ \ \ \ \ \ \ \ \forall \sigma_3.\, \sigma_2 \sqsubset \sigma_3 \sqsubseteq \sigma_1 \Rightarrow \textit{support}_{\langle X,\sigma_3\rangle}(P_1) = \emptyset) \\
&\ \ \ \ \ \ \ \ \vee \big(\forall \sigma_2 \sqsubseteq \sigma_1.\, \text{nonempty}(\sigma_2) \Rightarrow \textit{support}_{\langle X,\sigma_2\rangle}(P_1) = \emptyset\big)
\end{aligned}$$

The proof consists of constructing σ_2 and S' for all cases, given σ_1. When $\sigma_2 \sqsubset \sigma_1$, we also prove that σ_2 is maximal (*i.e.* there exists no σ_3).

$$\begin{array}{l}
|\sigma_1(y)| > 1 \Rightarrow \\
\quad S' = \{\langle k+1, min(\sigma_1(y))\rangle, \langle k+1, max(\sigma_1(y))\rangle\} \wedge \sigma_2 = \sigma_1 \\
\sigma_1(y) = \{i\} \Rightarrow \\
\qquad \sigma_2(X[i]) = \sigma_1(z) \cap \sigma_1(X[i]) \\
\qquad \wedge\ (\forall x \in \langle X \cdot y \cdot z\rangle.\, x \neq X[i] \ \Rightarrow\ \sigma_2(x) = \sigma_1(x)) \\
\qquad \wedge\ S' = \bigcup_{b \in \sigma_2(X[a])} \{\langle k+2, b\rangle\}
\end{array}$$

For the second case above, it remains to be shown that σ_2 is nonempty and maximal. We prove that σ_2 is maximal. For all values $b \in \sigma_2(X[a])$, a supporting literal $\langle z, b\rangle$ is required in S'. Therefore, P_1 implies that $\sigma_2(X[a]) \subseteq \sigma_2(z)$, hence $\sigma_2(X[a]) = \sigma_1(z) \cap \sigma_1(X[a])$ is maximal. For all other variables w, $\sigma_2(w) = \sigma_1(w)$, therefore σ_2 is maximal under $\sqsubseteq$.

If $\sigma_2(X[i]) = \emptyset$ (*i.e.* $\sigma_1(z) \cap \sigma_1(X[i]) = \emptyset$), σ_2 is empty. Since σ_2 is the maximal one which satisfies P_1, the second disjunct of the consequent of the schema holds.

Theorem 5 **(P_2 Support Generation)** *We consider P_2 on constraint* `Element`(X, y, z). *We claim that Def. 19 (propagation schema) holds for P_2.*

Proof Let $k = |X|$, and σ_1 and σ_2 be signatures mapping the variables in $X.y.z$ to their respective domains. The proof is by constructing σ_2 and S' to satisfy the first disjunct of the consequent of the schema.

$$\begin{array}{rl}
& \sigma_2(y) = \{i \in \sigma_1(y) \mid \exists a \in \sigma_1(z).\, a \in \sigma_1(X[i])\} \\
\forall x \in \langle X.z\rangle & \sigma_2(x) = \sigma_1(x) \\
& S' = \bigcup_{i \in \sigma_2(y)} \{\langle i, a\rangle, \langle k+2, a\rangle\}
\end{array}$$

σ_2 is maximal: the constructed σ_2 is identical to σ_1 except for the set $\sigma_2(y)$. For each value i of $\sigma_2(y)$, P_2 requires that there exists a value a in the domains of $X[i]$ and z. $\sigma_2(y)$ is the maximal subset of $\sigma_1(y)$ which satisfies this condition, therefore σ_2 is maximal under $\sqsubseteq$.

If σ_2 is empty, then (since σ_2 is maximal) the second disjunct of the consequent of the schema holds.

Theorem 6 **(P_3 Support Generation)** *We consider P_3 on constraint* `Element`(X, y, z). *We claim that Def. 19 (propagation schema) holds for P_3.*

Proof Let $k = |X|$, and σ_1 and σ_2 be signatures mapping the variables in $X.y.z$ to their respective domains. The proof is by constructing σ_2 and S' to satisfy the first disjunct of the consequent of the schema.

$$\begin{array}{rl}
& \sigma_2(z) = \{a \in \sigma_1(z) \mid \exists i \in \sigma_1(y).\, a \in \sigma_1(X[i])\} \\
\forall x \in X.y. & \sigma_2(x) = \sigma_1(x) \\
& S' = \bigcup_{a \in \sigma_2(z)} \{\langle i, a\rangle, \langle k+1, i\rangle\}
\end{array}$$

The constructed σ_2 is identical to σ_1 except for the set $\sigma_2(z)$. For each value a of $\sigma_2(z)$, P_3 requires that there exists an index i such that $a \in \sigma_2(X[i])$ and $i \in \sigma_2(y)$. $\sigma_2(z)$ is the maximal subset of $\sigma_1(y)$ which satisfies this condition, therefore σ_2 is maximal under $\sqsubseteq$.

If σ_2 is empty, then (since σ_2 is maximal) the second disjunct of the consequent of the schema holds.

4.1.4 Soundness and Completeness

Now we prove that the conjunction of the element support properties (Def. 21) is sound and complete using the semantics of element (Def. 20). We will write P_e for the set $\{P_1, P_2, P_3\}$.

Lemma 16 [P_e **is sound**]

$$\begin{array}{l}\forall\sigma.\ \text{singleton}(\sigma) \Rightarrow \\ \quad (\mathit{support}_{\langle X,\sigma\rangle}(P_e) \Rightarrow [\![\mathtt{element}(X,y,z)]\!]_\sigma \neq \emptyset)\end{array}$$

Proof Let σ be an arbitrary singleton signature. Since σ is a singleton it encodes a single tuple (say τ). Assume $\mathit{support}_{\langle X,\sigma\rangle}(P_e)$ holds. That is, supports for $P_1[\sigma]$, $P_2[\sigma]$ and $P_3[\sigma]$ are non empty. Now, consider P_1. Since $|\sigma(y)| = 1$ we know the first disjunct can not hold and so we must have support for the second. Since $\sigma(y) \neq \emptyset$ we know that there is a single tuple supporting the second disjunct of P_1 and since $|\sigma(X[i])| = 1$, to support P_1, τ must have the form $\langle x_1, \cdots, x_{i-1}, a, x_{i+1}, \cdots, x_k, i, a\rangle$. This same tuple supports P_2 and P_3. This tuple is clearly in $[\![\mathtt{element}(X,y,z)]\!]_\sigma$ and so soundness holds.

Theorem 7 [$\{P_1\}$ **complete**]

$$\begin{array}{l}\forall\sigma.\ [\![\mathtt{element}(\mathtt{X},\mathtt{y},\mathtt{z})]\!]_\sigma \neq \emptyset\ \Rightarrow \\ \quad \exists\sigma' \sqsubseteq \sigma.\ [\![\mathtt{element}(\mathtt{X},\mathtt{y},\mathtt{z})]\!]_{\sigma'} = [\![\mathtt{element}(\mathtt{X},\mathtt{y},\mathtt{z})]\!]_\sigma \\ \qquad \wedge\ \mathit{support}_{\langle Y,\sigma'\rangle}(\{P_1\})\end{array}$$

Proof Assume $[\![\mathtt{element}(X,y,z)]\!]_\sigma \neq \emptyset$ for arbitrary σ. If $\mathit{support}_{\langle Y,\sigma\rangle}(P_1) \neq \emptyset$ then the theorem is trivially true, so we assume that $\mathit{support}_{\langle Y,\sigma\rangle}(P_1) = \emptyset$ and construct a signature σ' that does not eliminate any solutions from the constraint and which is supports P_1.

The first disjunct of P_1 is supported whenever $|\sigma(y)| > 1$ and so if P_1 is not supported $\sigma(y) = \{i\}$ or $\sigma(y) = \emptyset$; by assumption no domain of σ is empty and so $\sigma(y) = \{i\}$. To falsify the second disjunct of P_1 when $\sigma(y) = \{i\}$, there must be some $a \in \sigma(X[i])$ such that the literal $\langle k+2, a\rangle$ can not be supported. This happens for any $a \in \sigma(X[i])$ where $a \notin \sigma(X[k+2])$. Let σ_1 be a signature that is just like σ except that

$$\sigma_1(X[k+2]) = \sigma(X[k+2]) \cap \sigma(X[i])$$

Since the constraint is non-empty the intersection is non-empty. The second disjunct of P_1 supports this new signature so it supports P_1. Clearly $\sigma_1 \sqsubseteq \sigma$ and so it only remains to show that the meaning of the constraint does not change under the new signature. It is enough to show that

$$[\![\mathtt{element}(X,y,z)]\!]_\sigma \subseteq [\![\mathtt{element}(X,y,z)]\!]_{\sigma_1}$$

Assume $\tau \in [\![\mathtt{element}(X,y,z)]\!]_\sigma$. Then $\tau \in X\text{–}\mathit{tuple}_\sigma$ is coherent and is of the form

$$\tau = \langle x_1, \cdots, x_{i-1}, a, x_{i+1}, \cdots, x_k, i, a\rangle$$

Since τ is an $X\text{–}\mathit{tuple}_\sigma$, we know $\tau[j] \in \sigma(X[j])$ for all $j \in \{1..k+2\}$. To construct σ_1 we simply eliminated elements $b \in \sigma(X[k+2])$ such that $b \notin \sigma(X[i])$ so since $a \in \sigma(X[i])$, $a \in \sigma_1(X[i])$ and $a \in \sigma_1(X[k+2])$ and so $\tau \in [\![\mathtt{element}(X,y,z)]\!]_{\sigma_1}$.

Theorem 8 [$\{P_2\}$ **complete**]

$$\begin{aligned}&\forall\sigma.\ [\![\texttt{element(X,y,z)}]\!]_\sigma \neq \emptyset \ \Rightarrow\\ &\quad \exists\sigma' \sqsubseteq \sigma.\ [\![\texttt{element(X,y,z)}]\!]_{\sigma'} = [\![\texttt{element(X,y,z)}]\!]_\sigma\\ &\qquad \wedge\ \mathit{support}_{\langle Y,\sigma'\rangle}(\{P_2\})\end{aligned}$$

Proof Note that if $P_2[\sigma]$ is unsupported then $\sigma(X[i]) \cap \sigma(z) = \emptyset$. But since we assume $[\![\texttt{element(X,y,z)}]\!]_\sigma \neq \emptyset$, this is impossible and so $P_2[\sigma]$ must be supported and completeness trivially holds.

Theorem 9 [$\{P_3\}$ **complete**]

Proof If there is no support for $P_3[\sigma]$ then

$$\exists a \in \sigma(z).\ \forall i \in \sigma(y).\ a \notin \sigma(X[i])$$

Just let σ' be the same as σ except that we remove all such elements from the domain of z in σ'.

$$\sigma'(z) = \sigma(z) \cap \bigcup_{i\in\sigma(y)} \sigma(X[i])$$

Clearly $\sigma'(z) \subset \sigma(z)$. The elements that have been removed could not be included in an solution of $[\![\texttt{element(X,y,z)}]\!]_\sigma$ and so we have lost no answers. Thus, we have shown P_3 is complete.

Corollary 5 [P_e **is complete**]

Proof The completeness of P_e follows from local completeness (Thm. 1) and the completeness of P_1, P_2 and P_3.

4.1.5 Discussion

The propagators derived here to enforce GAC on the element constraint are not identical to those presented by Gent et al. [15]. However they do follow the same general scheme. The main difference is that the propagators here use dynamic literal triggers in place of watched literals and a static assignment trigger. The concept of generalized support has allowed us to create these propagators within one formal framework.

4.2 New Watched Literal Propagators for Occurrence Constraints

The `occurrenceleq`(X, a, c) and `occurrencegeq`(X, a, c) constraints (very similar to `atmost` and `atleast`) restrict the number of occurrences of a value in a vector of variables. If $\mathrm{occ}(X, a)$ is the occurrences of value a in X, `occurrenceleq` states that $\mathrm{occ}(X, a) \leq c$ and `occurrencegeq` states that $\mathrm{occ}(X, a) \geq c$.

Occurrence constraints arise in many problems. For example, in a round-robin tournament schedule, it may be required that no team plays more than twice at each stadium, represented by occurrenceleq constraints. In car sequencing (car factory scheduling), occurrence constraints may be used to avoid placing too much demand on a work-station. Both problems are described on CSPLib [19].

First we present the formal semantics of occurrenceleq and occurrencegeq, followed by support properties for the two constraints.

Definition 22 **[Occurrenceleq Semantics]**

$$\begin{aligned}&[\![\texttt{occurrenceleq}(X,a,c)]\!]_\sigma = \langle X, R_X\rangle \quad \text{where}\\ &R_X = \{\ \tau \in X\textit{–tuple}_\sigma \mid \\ &\qquad\qquad |\{i \mid \tau[i]=a\}| \ \leq\ c \quad \}\end{aligned}$$

Definition 23 **[Occurrencegeq Semantics]**

$$\begin{aligned}&[\![\texttt{occurrencegeq}(X,a,c)]\!]_\sigma = \langle X, R_X\rangle \quad \text{where}\\ &R_X = \{\ \tau \in X\textit{–tuple}_\sigma \mid \\ &\qquad\qquad |\{i \mid \tau[i]=a\}| \ \geq\ c \quad \}\end{aligned}$$

4.2.1 Support Properties

Definition 24 **[Occurrence Support Properties]** Given a schema X, value a and occurrence count c, P_l is the support property for the `occurrenceleq` constraint, and P_g is the property for `occurrencegeq`.

$$\begin{aligned}P_l[\sigma](S) \stackrel{def}{=}\ & (\exists I \subseteq \{1\ldots|X|\}.\ |I| = (|X|-c+1)\wedge\\ & \forall i \in I.\exists b \neq a.\ \langle i,b\rangle \in S)\\ & \vee\\ & (\exists I \subseteq \{1\ldots|X|\}.\ |I| = (|X|-c)\ \wedge\\ & \forall i \in I.\ a \notin \sigma(X[i]))\end{aligned}$$

$$\begin{aligned}P_g[\sigma](S) \stackrel{def}{=}\ & (\exists I \subseteq \{1\ldots|X|\}.\ |I| = (c+1)\ \wedge\\ & \forall i \in I.\ \langle i,a\rangle \in S)\\ & \vee\\ & (\exists I \subseteq \{1\ldots|X|\}.\ |I| = c\ \wedge\\ & \forall i \in I.\ \ \nexists b \in \sigma(X[i]).\ b \neq a)\end{aligned}$$

P_g is slightly simpler, so we consider it first. There are two forms of support which can satisfy P_g, corresponding to the two disjuncts. The first disjunct can be satisfied if $c+1$ variables have a in their domain, by a support set which contains $c+1$ literals mapping distinct variables to a. The second disjunct is satisfied if c variables are *set* to a. In this case, S may be empty.

When it is no longer possible to satisfy the first disjunct, a corresponding propagator must narrow the domains to satisfy the second disjunct, by setting c variables to a. At this point, the constraint is trivially satisfied so S may be empty.

P_l is very similar, and essentially works in the same way except that it requires $|X|-c$ non-occurrences of a rather than c occurrences of a.

4.2.2 P-Admissibility and Backtrack Stability

We now prove that both properties meet the p-admissibility requirement.

Theorem 10 **[P_l P-Admissible]** *P_l is p-admissible according to Def. 12.*

Proof We case split on the disjuncts of P_l. The first disjunct does not refer to σ', and (since S has not changed) it remains true. The second disjunct is satisfied by $S=\emptyset$ only when the constraint is a tautology. Since $a \notin \sigma(X[i])$ and $\sigma' \sqsubseteq \sigma$, then $a \notin \sigma'(X[i])$ and the property remains true.

Theorem 11 [P_g **P-Admissible**] *P_g is p-admissible according to Def. 12.*

Proof We case split on the disjuncts of P_g. The first disjunct does not refer to σ', and (since S has not changed) it remains true. The second disjunct is satisfied by $S = \emptyset$ only when the constraint is a tautology. Since $\sigma(X[i]) \subseteq a$ and $\sigma' \sqsubseteq \sigma$, then $\sigma'(X[i]) \subseteq a$ and the property remains true.

In order for the two propagators to make use of watched literals, we must prove that both properties are backtrack stable. The watched literals representing a support are not backtracked, so a support must remain a support as search backtracks (and the domains are widened).

Theorem 12 [**Occurrence Backtrack Stable**] *The two occurrence support properties are backtrack stable according to Def. 13.*

Proof For both properties, the second disjunct is irrelevant because it is satisfied by $S = \emptyset$ only when the constraint is a tautology. The support $\emptyset$ is not required to be backtrack stable. In both properties the first disjunct requires a fixed number ($|X| - c + 1$ or $c + 1$) of literals to be in S (with variable indices I). It is clear that for any σ' where $\sigma \sqsubseteq \sigma'$, the same I may be used to discharge the existential, and S will be valid w.r.t σ'.

4.2.3 Proofs of the Propagation Schema

Now we give a constructive proof of the propagation schema for P_l. Recall that the computational content of the proof is a propagator for P_l.

Theorem 13 (**P_l Support Generation**) *We consider P_l on constraint* `occurrenceleq`(X, a, c). *We claim that Def. 19 (propagation schema) holds for P_l.*

Proof Let σ_1 and σ_2 be signatures mapping the variables in X to their respective domains. S and $\sigma_1 \sqsubseteq \sigma$ are universally quantified in the schema, therefore we use them as givens. We assume that $S \notin \mathit{support}_{\langle X, \sigma_1 \rangle}(P_l)$ and prove the consequent by constructing S' and σ_2. By lemma 12, $S \neq \emptyset$. The second disjunct of P_l would be satisfied by $S = \emptyset$, therefore S corresponds to the first disjunct of P_l.

S contains one literal for each index in I. At least one item in S is invalid (by the antecedant). The proof proceeds by constructing I' and corresponding S' and σ_2 to satisfy the first disjunct of P_l if possible. Otherwise, the second disjunct is satisfied by constructing σ_2 and $S' = \emptyset$.

$$\begin{aligned}
I_1 &= \{i \mid \langle i, b\rangle \in S \wedge (\exists b \neq a.\ b \in \sigma_1(X[i]))\}\\
I_2 &= \{i \mid i \notin I_1 \wedge (\exists b \neq a.\ b \in \sigma_1(X[i]))\}\\
I_3 &= I_1 \cup I_2
\end{aligned}$$

$$\begin{aligned}
|I_3| > (|X| - c) \Rightarrow\ & (I' \subseteq I_3 \wedge |I'| = (|X| - c + 1)\\
& \wedge S' = \{\langle i, b\rangle \mid i \in I'\\
& \qquad \wedge\ b \in \sigma_1(X[i])\ \wedge\ b \neq a\}\\
& \wedge \sigma_2 = \sigma_1)
\end{aligned}$$

$$\begin{aligned}
|I_3| = (|X| - c) \Rightarrow\ & S' = \emptyset\ \wedge\\
& (\forall i \notin I_3.\ \sigma_2(X[i]) = \sigma_1(X[i]))\ \wedge\\
& (\forall i \in I_3.\ \sigma_2(X[i]) = \sigma_1(X[i]) \setminus \{a\})
\end{aligned}$$

σ_2 is maximal in both of the above cases: in the first case, $\sigma_2 = \sigma_1$, and in the second case only the necessary values are removed to satisfy the second disjunct of P_l.

When $|I_3| < (|X| - c)$, P_l is false and remains false for all $\sigma_2 \sqsubseteq \sigma_1$ (by construction of I_1 and I_2). Hence the second disjunct of the consequent of the schema is satisfied.

The proof explicitly re-uses variable indices but not b values from S. This fits well with Minion's watched literal implementation, which notifies the propagator once for each invalid literal in S. However, the proof does not require the use of watched literals, it allows many concrete implementations and may be used with any propagation-based solver.

It is straightforward to prove the propagation schema for P_g, based on the proof for P_l.

Theorem 14 (P_g Support Generation) *We consider P_g on constraint* $\texttt{occurrencegeq}(X, a, c)$. *We claim that Def. 19 (propagation schema) holds for P_g.*

Proof The proof is the same as the proof of P_l, with c substituted for $|X| - c$ in all places, and $(a \in \sigma_1(X[i]))$ substituted for $(\exists b \neq a.\ b \in \sigma_1(X[i]))$, and $\{a\}$ substituted for $\sigma_1(X[i]) \setminus \{a\}$.

This proof also re-uses variable indices from S and thus fits well with Minion's watched literal infrastructure.

4.2.4 Soundness and Completeness

Now we prove the soundness and completeness of both properties, and hence the correctness of the two propagators.

Lemma 17 [Occurrenceleq Sound]

$$\begin{aligned}&\forall \sigma.\ \text{singleton}(\sigma) \Rightarrow\\ &\quad (\mathit{support}_{\langle X,\sigma\rangle}(P_l) \Rightarrow [\![\texttt{occurrenceleq}(X, a, c)]\!]_\sigma \neq \emptyset)\end{aligned}$$

Proof *Let σ be an arbitrary singleton signature. Since σ is a singleton it encodes a single tuple (say τ). Assume $support_{\langle X,\sigma\rangle}(P_l)$ holds. Let b be the number of occurrences of a in τ.*

Since σ is singleton, the first disjunct of P_l implies the second disjunct. (Assume I satisfies the first disjunct. $I' \subseteq I$ where $|I'| = (|X| - c)$ is used to satisfy the second disjunct.) Therefore $support_{\langle X,\sigma\rangle}(P_l)$ implies the second disjunct of P_l is satisfied (by the empty support). Hence, at least $|X| - c$ elements of τ are not equal to a, so $b \leq c$. By Def. 22, $R_X = \{\tau\}$ and the lemma holds.

The proof that P_g is sound proceeds by the same argument, with $|X| - c$ replaced with c, 'not equal to a' replaced with 'equal to a' and $\leq$ replaced with $\geq$.

Lemma 18 [Occurrenceleq Complete]

$$\begin{aligned}&C = \texttt{occurrenceleq}(X, a, c)\\ &\forall \sigma.\ [\![C]\!]_\sigma \neq \emptyset\ \Rightarrow\\ &\quad \exists \sigma' \sqsubseteq \sigma.\ [\![C]\!]_\sigma \subseteq [\![C]\!]_{\sigma'}\\ &\qquad \wedge\ \mathit{support}_{\langle X,\sigma'\rangle}(P_l)\end{aligned}$$

Proof Assume $[\![C]\!]_\sigma \neq \emptyset$ for arbitrary σ. If $\mathit{support}_{\langle X,\sigma\rangle}(P_l)$ then $\sigma' = \sigma$ and completeness trivially holds. Otherwise, by the proof of the propagation schema for P_l, there exists a $\sigma' \sqsubset \sigma$ (named σ_2 there) such that $\mathit{support}_{\langle X,\sigma'\rangle}(P_l)$. Since $\sigma' \neq \sigma$, σ' is constructed in the case where $|I_3| = (|X| - c)$. σ' is the same as σ except for indices I_3, where the value a is removed if present. For all $i \notin I_3$, $\sigma(i) = \{a\}$ therefore corresponding elements of all tuples $\tau \in [\![C]\!]_\sigma$ also equal a. No other element of τ can be a (by Def. 22), therefore no tuples are invalidated, $[\![C]\!]_{\sigma'} = [\![C]\!]_\sigma$ and the lemma holds.

Once again, the proof that P_g is complete follows the same argument. For P_g, $|I_3| = c$ and for all indices $i \in I_3$, $\sigma'(i) = \{a\}$. For other indices, the constructed σ' is equal to σ and does not contain a. By Def. 23, all tuples $\tau \in [\![C]\!]_\sigma$ must equal a at all indices I_3, therefore no tuples are invalidated under σ' and $[\![C]\!]_{\sigma'} = [\![C]\!]_\sigma$.

4.2.5 Empirical Evaluation

The occurrence propagators implemented in Minion 0.12 (and, to the best of our knowledge, all other solvers) use static triggers. Therefore they may be invoked when support has not been lost. By comparison, these watched literal propagators are only invoked when one of the literals in the support is lost.

We implemented the $\texttt{occurrenceleq}(X, a, c)$ propagator described by the proof of Theorem 13 in Minion 0.12. The propagator re-uses literals $\langle i, b\rangle$ from S when constructing S', allowing it to leave the corresponding watched literals in place. When a literal $\langle i, b\rangle$ in S is invalid, the propagator scans through $X[\{i..|X| - 1\}]$ then $X[\{0..i - 1\}]$ to find a replacement literal. The propagator (referred to as WatchedProp) was constructed from the proof in less than 3 hours programmer time.

We compare against the existing `occurrenceleq` propagator (StaticProp) provided in Minion 0.12, which uses static assignment triggers (*i.e.* the propagator is notified when any variable in scope becomes assigned).

We constructed a benchmark CSP as follows. We have a vector of variables X where $|X| = 100$, and initial signature σ where $\forall i.\ \sigma(X[i]) = 1, 2$. The constraints are as follows. $\forall i \in 80..98.\ (X[i] \neq X[i + 1])$, and 100 copies of the constraint $\texttt{occurrenceleq}(X, 1, 90)$. The occurrence constraint is duplicated to allow accurate measurement of its efficiency. This CSP is solved to find all solutions.

The solver branches on variables in X in index order, and branches for 1 before 2. Once variable $X[80]$ is assigned by search, the remaining variables are assigned by propagation on the $\neq$ constraints. As search progresses, the value of each variable in $X[\{80..99\}]$ alternates between 1 and 2.

WatchedProp watches 11 literals of the form $\langle i, 2\rangle$. Early in the search, most of these literals will necessarily involve variables $X[\{80..99\}]$, a pathological case for WatchedProp. As search progresses, more variables in $X[\{0..79\}]$ will be assigned 2, therefore the performance of WatchedProp should improve.

Table 1 shows that StaticProp scales approximately linearly in the number of search nodes explored, but WatchedProp speeds up as search progresses. With a limit of 100 million nodes, WatchedProp is more than twice as fast as StaticProp.

Search node limit (n)	WatchedProp time (s)	StaticProp time (s)
100,000	1.72	1.20
1,000,000	12.40	11.54
10,000,000	86.13	120.31
100,000,000	518.81	1205.07

Table 1 Times for the WatchedProp and StaticProp algorithms, median of 16 runs on a dual processor Intel Xeon E5520 at 2.27GHz.

5 Conclusions and Future Work

This paper has made a number of contributions to the formal study of constraint solving, in particular of propagation in constraint solving. We have shown that we can define formally a notion of generalized support, which generalizes the standard notion of support in constraint satisfaction. This generalization allows us to show that propagators which might not have been seen as using support. Since our definition is so general, we introduced the notion of "p-admissible" support properties. The definition of p-admissibility corresponds to the use of a particular kind of trigger within the constraint solver. Triggers are events which cause propagators to be called within the solver, and p-admissibility guarantees that any event which might cause support to be lost is observed by some trigger. In this paper we have focussed on a definition of p-admissibility corresponding to dynamic literal triggers. We have given a formal description of constraint propagation. Given a p-admissible support property, we have defined the propagation schema. A constructive proof of the propagation schema shows how a propagator can be constructed to find new support when support is lost. We have given examples of this for the specific constraints "element", "occurrenceleq" and "occurrencegeq".

Our work on propagators is not merely a formalisation of existing standard usage in constraint programming. We are not aware of a definition of support as general as ours within constraints. The notion of generalized support should be directly useful in constraints, enabling a much better understanding of propagation algorithms in the constraint community. Our hypothesis is that almost all propagators used in constraint solvers can be seen as reasoning with some form of support property, even though most propagators are not currently presented as doing so. Once this hypothesis is confirmed, we can present propagation algorithms in a much more uniform fashion, as well as building constraint solvers to exploit these propagation algorithms. Thus our intended future work consists of two strands: first continuing the formal development we have started here, and second demonstrating the application of our work to the constraints community.

Acknowledgements

The work of the authors has been partially supported by EPSRC grants EP/E030394/1, EP/F031114/1 and EP/H004092/1, support for which we are very grateful.

References

1. Krzysztof R. Apt. *Principles of Constraint Programming.* Cambridge University Press, 2003.
2. Krzysztof R. Apt and Eric Monfroy. Automatic generation of constraint propagation algorithms for small finite domains. In *Proceedings of the Fifth International Conference on Principles and Practice of Constraint Programming (CP 1999)*, pages 58–72, 1999.
3. Nicolas Beldiceanu, Mats Carlsson, Pierre Flener, María Andreína Francisco Rodríguez, and Justin Pearson. Linking prefixes and suffixes for constraints encoded using automata with accumulators. In *Proceedings of the 20th International Conference on Principles and Practice of Constraint Programming (CP 2014)*, pages 142–157, 2014.
4. Nicolas Beldiceanu, Mats Carlsson, and Thierry Petit. Deriving filtering algorithms from constraint checkers. In *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming (CP 2004)*, pages 107–122, 2004.
5. Christian Bessière. Constraint propagation. In P. Van Beek F. Rossi and T. Walsh, editors, *Handbook of Constraint Programming*, pages 29–83. Elsevier, 2006.
6. Christian Bessiere, Emmanuel Hebrard, Brahim Hnich, and Toby Walsh. The tractability of global constraints. In *Proceedings CP 2004*, pages 716–720, 2004.
7. Christian Bessière and Jean-Charles Régin. Arc consistency for general constraint networks: preliminary results. In *Proceedings 15th International Joint Conference on Artificial Intelligence (IJCAI 97)*, pages 398–404, 1997.
8. Christian Bessière and Jean-Charles Régin. Refining the basic constraint propagation algorithm. In *Proceedings 17th International Joint Conference on Artificial Intelligence (IJCAI 2001)*, pages 309–315, 2001.
9. James Caldwell, Ian Gent, and Judith Underwood. Search algorithms in type theory. *Theoretical Computer Science*, 232(1-2):55–90, Feb. 2000.
10. David Cohen, Christopher Jefferson, and Karen E. Petrie. When to watch and when to propagate. Under review.
11. Robert L. Constable. Naive computational type theory. In Helmut Schwichtenberg and Ralf Steinbruggen, editors, *Proof and System Reliability*, volume 62 of *Nato Science Series*, pages 213–259. Kluwer, 2002.
12. Robert L. Constable et. al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, 1986.
13. Eugene C. Freuder and Alan K. Mackworth. Constraint satisfaction: An emerging paradigm. In P. Van Beek F. Rossi and T. Walsh, editors, *Handbook of Constraint Programming*, pages 13–28. Elsevier, 2006.
14. Ian P. Gent, Chris Jefferson, and Ian Miguel. Minion: A fast, scalable, constraint solver. In *Proceedings 17th European Conference on Artificial Intelligence (ECAI 2006)*, pages 98–102, 2006.
15. Ian P. Gent, Chris Jefferson, and Ian Miguel. Watched literals for constraint propagation in minion. *Constraint Programming 2006*, LNCS 4204:182–197, 2006.
16. Ian P. Gent, Ian Miguel, and Peter Nightingale. Generalised arc consistency for the alldifferent constraint: An empirical survey. *Artificial Intelligence*, 172(18):1973–2000, 2008.
17. Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types.* Cambridge University Press, 1989.
18. Pascal Van Hentenryck, Helmut Simonis, and Mehmet Dincbas. Constraint satisfaction using constraint logic programming. *Artificial Intelligence*, 58:113–159, 1992.
19. Brahim Hnich, Ian Miguel, Ian P. Gent, and Toby Walsh. CSPLib: a problem library for constraints. http://csplib.org/.
20. *ILOG Solver 6.0 User Manual.* ILOG S.A., 2003.
21. Francoise Laburthe. Choco: a constraint programming kernel for solving combinatorial optimization problems. http://choco.sourceforge.net/.
22. Michael J. Maher. Propagation completeness of reactive constraints. In *Proceedings ICLP 2002*, pages 148–162, 2002.
23. J.-C. Régin. Generalized arc consistency for global cardinality constraint. In *Proc. AAAI 96*, pages 209–215, 1996.
24. Christian Schulte and Peter J. Stuckey. Efficient constraint propagation engines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(1), 2008.
25. Guido Tack, Christian Schulte, and Gert Smolka. Generating propagators for finite set constraints. *Constraint Programming 2006*, LNCS 4204:575–589, 2006.