Constraint Propagation and Explanation over Novel Types by Abstract Compilation*

Graeme Gange[1] and Peter J. Stuckey[2]

1 **Department of Computing and Information Systems, The University of Melbourne, Melbourne, Australia**
`gkgange@unimelb.edu.au`

2 **Data61, CSIRO and Department of Computing and Information Systems, The University of Melbourne, Melbourne, Australia**
`pstuckey@unimelb.edu.au`

Abstract

The appeal of constraint programming (CP) lies in compositionality – the ability to mix and match constraints as needed. However, this flexibility typically does not extend to the types of variables. Solvers usually support only a small set of pre-defined variable types, and extending this is not typically a simple exercise: not only must the solver engine be updated, but then the library of supported constraints must be re-implemented to support the new type.

In this paper, we attempt to ease this second step. We describe a system for automatically deriving a native-code implementation of a global constraint (over novel variable types) from a declarative specification, complete with the ability to explain its propagation, a requirement if we want to make use of modern lazy clause generation CP solvers.

We demonstrate this approach by adding support for *wrapped-integer* variables to `chuffed`, a lazy clause generation CP solver.

1998 ACM Subject Classification F.4.1 Mathematical Logic, I.2.2 Automatic Programming

Keywords and phrases constraint programming, program synthesis, program analysis

Digital Object Identifier 10.4230/OASIcs.ICLP.2016.13

1 Introduction

A large factor in the success of constraint programming (CP) is compositionality – the flexibility to freely mix and match constraints as needed. However, we are reliant on the underlying solver to provide efficient propagator implementations for the constraints of interest. If we require some problem-specific global constraint we must either design and implement bespoke propagation (and, if we want to use modern lazy clause generation solvers [17], explanation) algorithms or decompose our global constraint into supported primitives.

CP solvers typically support only integer, Boolean and occasionally set variables. Suppose we wish to solve problems over some other algebraic structure – a finite semiring, or the two's complement (or *wrapped*) integers. In this case, we need some way to represent variable domains, encode the semantics of operations, and provide implementations of all constraints of interest.

This can be done by representing variables with existing types and emulating constraints by decomposition into existing primitives. However, a decomposition into existing primitives

* This work was supported by Australian Research Council DE160100568.

```
lex_lt([X|_], [Y|_]) :- X < Y.
lex_lt([X|Xs], [Y|Ys]) :-
  X = Y, lex_lt(Xs, Ys).
```

(a)

```
lex_lt1(X1, X2, Y1, Y2) :- X1 < Y1. % c1
lex_lt1(X1, X2, Y1, Y2) :-          % c2
  X1 = Y1, lex_lt2(X2, Y2).
lex_lt2(X2, Y2) :- X2 < Y2.         % c3
```

(b)

Figure 1 Specification of a strict lexicographic order, and concrete instantiation on arrays of length 2.

```
Prop(lex_lt1)($s_0$):
    $s_1$ := $\mathcal{R}^{\#}$(v1 < v3)($s_0$)
    $s_2$ := $\mathcal{R}^{\#}$(v1 = v3)($s_0$)
    $s_3$ := $\textsc{Rename}(s_2, [(\texttt{v1} \leftarrow \texttt{v2}), (\texttt{v2} \leftarrow \texttt{v4})])$
    $s_4$ := $\mathcal{R}^{\#}$(v1 < v2)($s_3$)
    $s_5$ := $\textsc{Splice}(s_2, s_4, [(\texttt{v2} \leftarrow \texttt{v1}), (\texttt{v4} \leftarrow \texttt{v2})])$
    $s_6$ := $\mathcal{R}^{\#}$(v1 = v3)($s_5$)
    $s_7$ := $\textsc{Join}([s_1, s_6])$
    return $s_7$
```

```
Expl(lex_lt1)($[s_1, \ldots, s_7], e_0$):
    $e_1$ := $E_{\mathcal{R}^{\#}}$(v1 = v3)($e_0, s_5$)
    $e_2$ := $\textsc{ESplice}(e_1, s_2, s_4,$
                 $[v1 \leftarrow v2, v1 \leftarrow v4])$
    $e_3$ := $E_{\mathcal{R}^{\#}}$(v1 < v2)($e_2[2], s_3$)
    $e_4$ := $\textsc{ERename}(e_3, s_2,$
                 $[(\texttt{v2} \leftarrow \texttt{v1}), (\texttt{v4} \leftarrow \texttt{v2})])$
    $e_5$ := $\textsc{Meet}([e_4, e_2[1]])$
    $e_6$ := $E_{\mathcal{R}^{\#}}$(v1 = v3)($e_5, s_0$)
    $e_7$ := $E_{\mathcal{R}^{\#}}$(v1 < v3)($e_0, s_0$)
    $e_8$ := $\textsc{Meet}([e_7, e_6])$
    return $e_8$
```

Figure 2 Propagator and explanation computations derived for the constraint in Figure 1.

may be non-obvious and may be quite large. The decomposition may also be quite unwieldy, as customized decompositions must be provided for all global constraints of interest. Decomposition approaches may also sacrifice efficiency and propagation (and explanation) strength.

A more convenient (for the user) way of handling decomposition approaches is as a model transformer; an extended language is defined, supporting the new types of interest, and are compiled down to the core modelling language. This is the approach adopted for finite-extension [5] and option types [15]. Though convenient for modelling, this requires building a parser and compiler, in addition to the expressive limitations of decompositions.

The alternate approach is to integrate the new variable type natively into the solver. Native integration is typically a very substantial undertaking, so is rarely done.

In this paper, we develop a method for dynamically compiling native-code implementations of propagators from declarative specifications. We then use this to construct global propagators for integer variables with two's complement semantics. We have implemented the described approach as a standalone library, which we then integrated into the `chuffed` [6] lazy clause generation constraint programming solver.

The key insight of our approach is that propagation is a form of abstract interpretation, and hence we can use abstract compilation to generate implementations of propagators

▶ **Example 1.** Consider defining strict lexicographic inequality constraint. A possible checker for this constraint is shown in Figure 1(a). If we wish to instantiate a propagator for a particular constraint, we need to unfold the definition using the structure of the constraint. The unfolded definition for `lex_lt([X1,X2],[Y1,Y2])` is shown in Figure 1(b).

We build a propagator by computing approximations of, for each program point, the set of execution states which are reachable from the initial call, and the subset of those states which could succeed.

Figure 2(a) shows the generated propagator for the constraint `lex_lt([X1,X2],[Y1,Y2])`, we will discuss the detailed meaning later in the paper. Both clauses of `lex_lt1` are reachable from any initial state (s_0). Clause `c1` succeeds iff `v1 < v3` holds, so s_1 approximates its success set. For c2, s_2 approximates the set of states which may reach the call to lex_lt2, which is mapped onto the formal parameters in s_3. At s_4, we have computed the success set for lex_lt2. s_5 and s_6 then compute the corresponding success set for c2. s_7 combines the succeeding states for c1 and c2, returning newly pruned variable domains.

The explanation procedure given in Figure 2(b) simply retraces the computations performed by the propagator: for each instruction I with predecessor s_{pre} and necessary condition e_{post}, we compute some e_{pre} such that $s_{pre} \sqsubseteq e_{pre}$, and $I^{\#}(e_{pre}) \sqsubseteq e_{post}$. ◀

The contributions of this paper are as follows:

- A high-level declarative language for specifying constraints
- A procedure for partial evaluation of this high-level language down to a simple constraint logic programming language
- A procedure for deriving abstract propagator and explanation algorithms from these constraint definitions
- A method for synthesizing concrete implementations from these abstract propagators and explainers over novel variable types.

In the following section, we give a brief overview of constraint propagation and abstract interpretation. In Section 3, we describe the correspondence between propagation and static analysis, then in Sections 4 and 5, we show how to use this correspondence to derive propagation and explanation algorithms from implementations of checkers. In Sections 6 and 7, we describe integration of these propagators into a solver, and deriving checkers from a more expressive declarative language. Finally, Section 8 gives an example application of this approach, we describe related work in Section 9 then conclude in Section 10.

2 Preliminaries

In this paper, we restrict ourselves to *finite domain* constraint satisfaction and optimization problems (CSPs and COPs). To avoid confusion, we shall denote logical implication with $\Rightarrow$, and the set of functions with $\rightarrow$.

Propagation-based constraint solving

A CSP is defined by a tuple $(\mathcal{V}, \mathcal{D}, \mathcal{C})$ consisting of a set of variables $\mathcal{V}$ where each variable v may take values from a fixed finite set $\mathcal{D}(v)$, and a set of constraints $\mathcal{C}$. A constraint $c \in \mathcal{C}$ has a *scope*, $scope(c)$ which is a set of variables in $\mathcal{V}$. A constraint c with $scope(c) = \{v_1, \ldots, v_n\}$ is a set of assignments mapping each $v_i \in scope(c)$ to a value in $\mathcal{D}(v_i)$. A *solution* to a CSP is an assignment to each $v \in V$ such that every constraint in C is satisfied. In an abuse of notation we say assignment $\theta \in \mathcal{D}$, if $\theta(v) \in \mathcal{D}(v)$ for all $v \in \mathcal{V}$. A domain is *singleton* if it represents a single assignment, e.g. $|\mathcal{D}(v)| = 1, v \in Vars$. We denote the valuation corresponding to a singleton domain $\mathcal{D}$ as $\theta_{\mathcal{D}}$.

A *propagator* f for a constraint c is a decreasing function, from domains to domains which eliminates values which are not part of any solution to c. A propagator is *correct* if it does not exclude any satisfying assignments – that is, $\theta \in c \wedge \theta \in \mathcal{D} \Rightarrow \theta \in f(c)(\mathcal{D})$. A propagator is *checking* if it is exact for singleton domains, i.e. $f(\mathcal{D}) = \mathcal{D}$ for singleton domains iff $\theta_{\mathcal{D}} \in c$.

In a nogood-learning/lazy clause generation [17] solver, inferences/domain reductions are couched in terms of a formal language of *atomic constraints*, which form a complemented, partially ordered set. A common example in finite-domain solvers is the language of integer bounds and (dis-)equalities: $\{\langle x \leq k \rangle, \langle x > k \rangle, \langle x = k \rangle, \langle x \neq k \rangle\}$, for some variable x and integer constant k. Where a solver integrates nogood-learning/lazy clause generation [17] techniques, each inference inf resulting from a propagation $f(c)(D)$ is associated with a corresponding *explanation* E. E is a conjunction of atomic constraints such that $D \Rightarrow E$ and $c \wedge E \Rightarrow inf$. The first condition ensures E is true under the current state, and the second ensures $E \Rightarrow inf$ is globally valid in the problem. When a conflict is detected, inferences participating in the conflict are successively replaced by their explanations to derive a valid nogood which eliminates the current branch of the search tree.

Static program analysis by abstract interpretation

The construction of our propagation and explanation algorithms will be based on the machinery of program analysis.

Abstract interpretation [7] is a framework for inferring information about the behaviour of a program by performing computation on an *abstraction* of the program. The domain $\mathcal{A}$ of program states is replaced by an *abstraction* $\mathcal{A}^{\#}$. The abstract domain $\mathcal{A}^{\#}$ forms a lattice, equipped with the usual operators $(\sqsubseteq, \sqcup, \sqcap)$. Correspondence between concrete and abstract states is established by a pair (α, γ) of an abstraction and a concretization function, which form a Galois connection.

Each program instruction $\mathcal{T} : \mathcal{A} \rightarrow \mathcal{A}$ is similarly replaced with an abstraction $\mathcal{T}^{\#} : \mathcal{A}^{\#} \rightarrow \mathcal{A}^{\#}$. Properties of the program are inferred by executing this abstracted program. Rather than directly executing (possibly infinitely many) control paths, abstract interpreters typically store a single approximation of each program point. Where multiple control paths merge (after conditional statements, at loop heads, or function entries), the incoming abstract states are instead combined: $\varphi_p = \bigsqcup_{q \in \mathsf{preds}(p)} \varphi_q$. Thus, φ_p consists of the strongest property (representable in $\mathcal{A}^{\#}$) which holds in all predecessor states. This avoids the so-called *path-explosion* problem, but sacrifices precision at join points. Starting with all program points (except the entry) unreachable, state transformers are repeatedly evaluated until a fixpoint is reached. If each transformer is a sound overapproximation, any property which holds at the fixpoint also holds in any reachable concrete state. A typical application of this is to infer numerical properties which must hold at each program point. This is a so-called *forward analysis*, as properties at a given program point are derived from its predecessors.

In a backwards analysis, properties of states are derived from their successors. Numerical backwards analyses are typically rarer than forward analyses. In this case, it is important to distinguish *necessary* preconditions, which must hold in *any* predecessor of a given state, from *sufficient* conditions, which guarantee the given property will hold. Inference of *necessary* conditions have been used to infer preconditions from assertions [9].

To perform a backward analysis in the abstract interpretation framework, we must construct state transformers under/over-approximating the pre-image $\mathcal{T}^{-}$ of program statements. The analysis proceeds in a similar manner to the forward analysis, but proceeds backwards along the flow of execution, replacing the abstract transformer $\mathcal{T}^{\#}$ with an abstraction $\mathcal{T}^{-\#}$ of the pre-image.

In this paper, we shall require both forms of analysis; forwards to compute reachable states, and backwards to determine which of these satisfy a constraint.

3 Propagation as Static Analysis

Consider some constraint c, and a checker *program* $CH(c)$ which maps valuations θ over $scope(c)$ to true/false such that $CH(c)(\theta) = \mathsf{true} \Leftrightarrow c(\theta)$. The semantics of c is exactly the set of assignments θ such that *executing* $CH(c)$ returns `true` (from a logic programming

$$\begin{array}{rcl}
\tau & \rightarrow & \texttt{ident} \mid \texttt{const} \mid \texttt{ident}(\tau^*) \\
\alpha & \rightarrow & \texttt{ident} \mid \texttt{const} \mid \texttt{ident}(\alpha^*) \\
 & & \mid \alpha \otimes \alpha \mid \ominus \alpha \\
\mathit{def} & \rightarrow & \texttt{ident} := \alpha \\
\mathit{guard} & \rightarrow & \alpha \ \mathit{op} \ \alpha, \ \mathit{op} \in \{=, \neq, <, \leq\} \\
\mathit{call} & \rightarrow & \texttt{ident}(\tau^*) \\
\mathit{stmt} & \rightarrow & \mathit{def} \mid \mathit{guard} \mid \mathit{call} \\
\mathit{clause} & \rightarrow & \texttt{ident}(\tau^*) \ \texttt{:-} \ \mathit{stmt}^*
\end{array}$$

(a)

$$\begin{array}{rcl}
\tau & \rightarrow & \texttt{ident} \mid \texttt{const} \\
\alpha & \rightarrow & \texttt{ident}(\tau^*) \mid \tau \otimes \tau \mid \ominus \tau \\
\mathit{def} & \rightarrow & \texttt{ident} := \alpha \\
\mathit{guard} & \rightarrow & \tau \ \mathit{op} \ \tau, \ \mathit{op} \in \{=, \neq, <, \leq\} \\
\mathit{call} & \rightarrow & \texttt{ident}(\tau^*) \\
\mathit{stmt} & \rightarrow & \mathit{def} \mid \mathit{guard} \mid \mathit{call} \\
\mathit{clause} & \rightarrow & \texttt{ident}(\tau^*) \ \texttt{:-} \ \mathit{stmt}^*
\end{array}$$

(b)

Figure 3 (a) A LP-style specification language $\mathcal{L}$ for constraints, and (b) The simplified intermediate language $\mathcal{L}^-$, having eliminated complex terms, expressions and recursion. $\otimes$ is a binary infix arithmetic operator, $\ominus$ is a unary arithmetic operator.

perspective, this is the set of *answers* of $CH(c)$). Indeed, any backwards reachability analysis (from `true`) on $CH(c)$ computes a sound approximation of c.

If we interpret the solver's domain store $\mathcal{D}$ as an abstraction of assignments, then a *propagator* $P(c)$ is simply an approximation of the answers of $CH(c)$ restricted to $\gamma(\mathcal{D})$. This is, in fact, equivalent to the *contract precondition inference problem* described in [9] – given a transition system (the program) and initial states (the domain), find the strongest properties which eliminate *only* bad states.

Not every analysis is a valid propagator, however. Propagators will be called on a complete assignment to verify that the assignment is a solution. Each propagator must therefore be *checking* to ensure soundness. This is an extremely uncommon property for a general numeric analysis to have – even from a concrete initial state, precision may be lost at join points, and *widening* [8] discards properties to ensure termination in the presence of unbounded loops.

Nevertheless, this gives us the rough skeleton of an approach: given some specification of a constraint and suitable implementations of abstract operations, we shall generate a native-code implementation of an answer-set analysis for the specification.

But first, we must choose the manner of our specifications.

3.1 Programs as Constraints

While this derivation of propagators from programs is *possible* for arbitrary source languages, in practice we must consider both ease of specification (from the user's perspective) and effectiveness of analysis.

For the remainder of this paper, we consider specifications given in a small (C)LP-style language, $\mathcal{L}$, shown in Figure 3(a). The syntactic category τ denotes the usual language of *terms*. α is the syntactic category of arithmetic expressions, which will be eagerly evaluated during execution. We impose two additional syntactic restrictions. First, free variables cannot be introduced in clause bodies. Second, all (possibly indirect) recursion must be structurally decreasing. That is, if some call `p(X')` is reachable from a call `p(X)`, `X'` must be strictly smaller than `X` with respect to some well-founded measure on term *structure* (independent of the values of variables/constants).

This language $\mathcal{L}$ is reasonably expressive, and provides natural formulations for many global constraints, but does not necessarily seem amenable to numeric analysis.

However, the first condition above ensures that all computations are performed on ground values – this will be needed to ensure the propagators correctly reject invalid total assignments. The second condition similarly guarantees that recursion can be statically expanded. When a constraint is instantiated, we can partially evaluate the specification to construct a much simpler acyclic program consisting only of primitive guards, definitions and calls, which we shall use to derive our propagators.

The reduced language $\mathcal{L}^-$ is shown in Figure 3(b), which eliminates structured terms, complex expressions and all functions (except primitive operators and guards).

4 Constructing propagators from programs

Given a program in the intermediate language $\mathcal{L}^-$ described in Section 3, we must construct a program which, for a given input domain, computes an overapproximation of the corresponding concrete inputs which succeed. To do so, we first construct an intermediate representation of the computations performed by the propagator.

Propagator operations

The instructions used in the constructed propagators: postcondition $\mathcal{T}^{\#}(\texttt{stmt})(q)$, precondition $\mathcal{T}^{-\#}(\texttt{stmt})(q)$, relation $\mathcal{R}^{\#}(\texttt{rel})(q)$, disjunction $\textsc{Join}([q_1, \ldots, q_n])$, conjunction $\textsc{Meet}([q_1, \ldots, q_n])$, projection $\textsc{Rename}(q, [y_1 \leftarrow x_1, \ldots, y_1 \leftarrow x_n])$, and partial update $\textsc{Splice}(q, q', [y_1 \leftarrow x_1, \ldots, y_n \leftarrow x_n])$. Each operation computes an approximation of execution states from one or more previous states. $\textsc{Join}$ and $\textsc{Meet}$ respectively compute the least upper bound ($\sqcup$) and greatest lower bound ($\sqcap$) of abstract states under $\mathcal{A}^{\#}$. $\mathcal{T}^{\#}$ and $\mathcal{T}^{-\#}$ are respectively post- and precondition transformers for function applications, and $\mathcal{R}^{\#}$ applies a relation to an existing state. The remaining operations, are used in dealing with predicate calls. $\textsc{Rename}$ maps variables at a call site onto the formal parameters of the callee. $\textsc{Splice}$ copies a given state q, but takes the domains of variables $v_1, \ldots$ from some *other* state q'. This is used to weave the results of a call back into state of the caller.

The execution of some clause c operates on an execution environment E mapping names to constants. Each guard evaluates the current context, and execution fails if the constraint is violated. A definition adds a new binding to the current environment. At each call site, we rename the call parameters and execute the predicate with the resulting environment. For predicates, each clause is simply executed in turn under the current environment, until some clause succeeds. If all clauses fail, the predicate likewise fails.

Analysis of c simply mirrors the program execution. From an abstract state Q_c, we compute approximations of the reachable states after executing each statement A_c. After computing the abstract solutions of predicate calls, we apply inverse state transformers to determine which initial environments correspond to the solutions. For predicates, the analysis is straightforward: compute the solution sets $[A_{c_1}, \ldots, A_{c_k}]$ for each clause $[c_1, \ldots, c_k]$, and compute the abstract join of these, so $A_p = A_{c_1} \sqcup \ldots \sqcup A_{c_k}$. Throughout the analysis, we maintain the property that $Q_i \sqsubseteq A_i$ – every ‘solution’ is (abstractly) reachable. This is relatively easy to preserve for guards (which are descending) and definitions. As definitions are total functions, executing some definition `x := E` only introduces a new binding `x`. For any state φ, we then have $\exists x.\ \mathcal{T}^{\#}(\texttt{x := E})(\varphi) = \varphi$. Thus, even the trivial pre-image computation $\mathcal{T}^{-\#}(\texttt{x := E})(\varphi) = \exists\ x.\ \varphi$ preserves this invariant. The upshot of this is that we need not explicitly compute $A_i = A_i \sqcap Q_i$, as this is naturally preserved.

We run into some complications at the predicate level, however. As mentioned in Section 2, abstract interpreters perform abstract *join* operations ($\sqcup$) to combine states whenever a program point is reachable along multiple control paths. If `p` is called in several contexts, if we directly retrieve the solutions to `p` at the call-site, we may lose the property that

Query computation	

`p(x1, ...) :- c1; ...; ck`	$s =$ push_state(JOIN(retrieve_callers(p)))
	save_clause(c1, $Q(s \mid$ c1))
	. . .
	save_clause(c1, $Q(s \mid$ ck))

$Q(s \mid \emptyset)$	s
$Q(s \mid$ `x := E`$, c)$	$Q($push_state($\mathcal{T}^{\#}$(`x := E`)$(s)) \mid c)$
$Q(s \mid$ `x op y`$, c)$	$Q($push_state($\mathcal{R}^{\#}$(`x op y`)$(s)) \mid c)$
$Q(s \mid$ `p(x1, ...)`$, c)$	save_call(p, push_state(RENAME(s, [x1, ...]))); s
Answer computation	
`p(x1, ...) :- c1; ...; ck`	$s_1 = A($retrieve_clause(c1) $\mid$ c1) . . . $s_k = A($retrieve_clause(ck) $\mid$ ck) $s =$ push_state(JOIN)($[s_1, \ldots, s_k]$) save_answer(p, s)
$A(s_0 \mid \emptyset)$	s_0
$A(s_0 \mid$ `x := E`$, c)$	push_state($\mathcal{T}^{-\#}$(`x := E`)$(A(s_0 \mid c))$)
$A(s_0 \mid$ `x op y`$, c)$	push_state($\mathcal{R}^{\#}$(`x op y`)$(A(s_0 \mid c))$)
$A(s_0 \mid$ p($\mathtt{x_1}, \ldots, \mathtt{x_n}$)$, c)$	$s_{post} = A(s_0 \mid c)$ $s_{proj} =$ push_state(RENAME(s_{post}, [x1, ...])) $s_{ret} =$ retrieve_answer(p) $s_{meet} =$ push_state(MEET($[s_{ret}, s_{proj}]$) push_state(SPLICE($s_{post}, s_{meet}, [\mathtt{x_1}, \ldots, \mathtt{x_n}]$))

Figure 4 Computing approximations of reachable and satisfying states during checker execution. Reachability computations are performed for predicates in topological order, and answers are computed in the reverse order.

$A_i \sqsubseteq Q_i$. Worse, the loss of precision can interfere with the requirement that the propagator be checking.

▶ **Example 2.** Consider the following program:

```
p(x, y) :- q(x, y).     p(x, y) :- q(y, x).     q(u, v) :- u = v.
```

Consider analysing this program under $\{\mathtt{x} \rightarrow 3, \mathtt{y} \rightarrow 4\}$. `q` is reachable under two environments: $\{\mathtt{u} \rightarrow 3, \mathtt{v} \rightarrow 4\}$, and $\{\mathtt{u} \rightarrow 4, \mathtt{v} \rightarrow 3\}$. Before processing `q`, the calling contexts are combined into $\{\mathtt{u} \rightarrow [3,4], \mathtt{v} \rightarrow [3,4]\}$. Applying `u = v` here does nothing. Notice that the answer set of `q` is weaker than either call state. When we combine this back into the call site, both calls appear feasible, so we do not detect failure. ◀

To preserve the descending property, we must instead compute the meet of the calling state with the answer set of the predicate. To ensure the propagator is checking, we exploit the fact that *bindings* are functionally defined. We transform the *checker* to ensure all calls to a predicate to have identical argument definitions (in terms of input variables). We can perform this step by traversing the program tracking the definition of each variable, and renaming apart predicate calls with different definitions. In the case of Example 2, `q` becomes two separate predicates `q1` and `q2`.

The algorithm for constructing a propagator from a checker is given in Figure 4. push_state adds a new state to the propagator, and returns the new state's identifier. In addition to the generated instructions, we also need to keep track of three sets of states: states which call some predicate `p`, the final state of each clause `c`, and the answer set of each predicate `p` – we

Instruction	Generated code	Resulting state
$\mathcal{T}^{\#}(\texttt{z := f(x,y)})(c,\sigma)$	$v' := \mathit{emit}(\mathcal{T}^{\#}(f)(\sigma(x),\sigma(y)))$	$(c,\sigma[z\mapsto v'])$
$\mathcal{T}^{-\#}(\texttt{z := f(x,y)})(c,\sigma)$	$b,u',v' :=$ $\quad \mathit{emit}(\mathcal{T}^{-\#}(f)(\sigma(z),\sigma(x),\sigma(y)))$ $c' := c \wedge b$	$(c,\sigma[x\mapsto u', y\mapsto v']\setminus\{z\})$
$\mathcal{R}^{\#}(\texttt{x rel y})$	$b,u',v' := \mathit{emit}(\mathcal{R}^{\#}(\texttt{rel},\sigma(x),\sigma(y)))$ $c' := c \wedge b$	$(c',\sigma[x\mapsto u', y\mapsto v'])$
$\textsc{Meet}([(c_A,\sigma_A),(c_B,\sigma_B)])$	$b_x, v_x := \sigma_A(x) \sqcap \sigma_B(x)$ **for** $x \in \sigma_A$ $c' := c_A \wedge c_B \wedge \bigwedge b_x$ $\sigma' := \{x \mapsto v_x \mid x \in \sigma_A\}$	(c',σ')
$\textsc{Join}([(c_A,\sigma_A),(c_B,\sigma_B)])$	$v_x := \begin{pmatrix} \sigma_B(x) \textbf{ if } \neg c_A \\ \sigma_A(x) \textbf{ if } \neg c_B \\ \sigma_A(x) \sqcup \sigma_B(x) \textbf{ else} \end{pmatrix}$ **for** $x \in \sigma_A$ $c' := c_A \vee c_B$ $\sigma' := \{x \mapsto v_x \mid x \in \sigma_A\}$	(c',σ')
$\textsc{Rename}((c,\sigma),\ M)$		$(c, \{y \mapsto \sigma(x)$ $\mid (y \leftarrow x) \in M\})$
$\textsc{Splice}((c,\sigma),(c_{sp},\sigma_{sp})$,M)		$(c_{sp}, \sigma[y \mapsto \sigma_{sp}(x)$ $\mid (y \leftarrow x) \in M])$

Figure 5 Constructing concrete code implementing an abstract propagator. *emit*() denotes dispatch to an externally-provided transfer function.

use the corresponding save/retrieve functions to keep track of these sets. $Q(s \mid c)$ constructs the computation of final reachable states of clause c starting from state s, and records the context of any predicate calls made.

4.1 Generating Propagator Implementations

The propagator descriptions described above make no assumptions as to the concrete representation of propagator states, other than being elements of a lattice with associated state transformers. For the remainder of the paper, we shall assume states are abstracted by a non-relational ('independent attribute') domain. The concrete representation of a state is then a tuple (c,σ), where c is a Boolean flag indicating whether the state is feasible, and σ is a mapping from variables to the corresponding domain representation.

Under this non-relational representation, generating concrete implementations of these propagators is relatively straightforward. Propagator computation consists of three phases: a prologue, where domain representations are extracted from solver variables, the propagator body, and an epilogue, where we compare the initial and revised domains for each variable, and post any updated domains to the solver. The propagator body simply computes values for the sequence of states appearing in the abstract propagator we constructed. State transformers for operations on individual domain approximations must be externally provided, which we then lift to operations on propagator states.

Rules for state computation are given in Figure 5. We assume machine code is written to a global buffer. In the generated code, RENAME and SPLICE become no-ops; they simply re-bind existing values to new names. $\mathcal{T}^{\#}$, $\mathcal{T}^{-\#}$ and $\mathcal{R}^{\#}$ are similarly straightforward, computing new values for those variables touched by the instruction (using the externally provided implementations), and updating the corresponding bindings. Here *emit* denotes calls to an external code generator, which emits instructions implementing the specified primitive, and returns the location of the resulting values. JOIN and MEET implement the usual lifting of $\sqcup$ and $\sqcap$ operations to the Cartesian product. We show here code only for binary functions, as well as meet and join; the n-ary operators follow the same pattern.

5 Inferring Explanations

In nogood-learning solvers, we have an additional complication: explanations. Assume a propagation step $f(c)(D)$ infers the atomic constraint at. During conflict analysis, we will need to replace at with some set of antecedents $l_1, \ldots, l_k$ such that $D \Rightarrow l_1 \wedge \ldots \wedge l_k$, and $c \wedge l_1 \wedge \ldots \wedge l_k \Rightarrow at$.

When it comes to dealing with novel variable types, we have two problems: first, how to represent atomic constraints in general, and how to infer explanations for arbitrary constraints, while avoiding imposing too heavy a burden on the solver author.

To this end, we make a pair of perhaps trivial observations. First, a variable domain is always expressible as a conjunction of atomic constraints. Second, the generated propagators always admit some valid explanation consisting of a conjunction of variable domains. This hints at a possible approach - collect explanations using the same domain representation as the propagation algorithm, and have the solver extract the corresponding atomic constraints before returning.

Note that we can't absolve the solver developer from integrating atomic constraints into the solver core; handling propagation, implication and resolution of atomic constraints is still something that needs to be done.[1] However, with this approach they do not need to somehow communicate the semantics of atoms to the synthesis engine, nor provide bindings for the full set of operations (subsumption, disjunction, etc.) on atoms.

In terms of generating the explanation itself, the trivial explanation is always sound, relatively efficient to construct and requires no additional information from the solver, but is of limited value: $\bigwedge\{\mathcal{D}(v) \mid v \in \mathit{scope}(c)\} \Rightarrow at$. We can do much better by taking the correspondence between static analysis and propagation one step further, and observe that explanation is just an analysis of $P(c)$. Recall the computation of $P(c)$, illustrated to the right. From some initial state D, we apply a sequence of state transformers $[T_1, \ldots, T_n]$ computing states $[D_1, \ldots, D_n]$, D_n being the approximate solution set.

For an inference D_{inf}, we wish to find D_{expl} such that $D \sqsubseteq D_{\mathit{expl}}$, and $P(c)(D_{\mathit{expl}}) \sqsubseteq D_{\mathit{inf}}$. We can compute such a state by pushing the condition backwards along the computation of $P(c)$. We first find some state E_{n-1}, with $D_{n-1} \sqsubseteq E_{n-1}$ and $T_n(E_{n-1}) \sqsubseteq D_{\mathit{inf}}$. We continue in this manner, at each step computing E_{i-1} from D_{i-1} and E_i. The final state, E_0 is thus guaranteed to be a valid explanation.

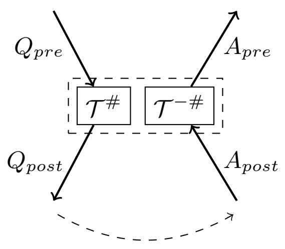

Figure 6 Flow of computation in $P(c)$.

[1] Though the developer may be able to re-use atoms for existing types – encoding option types with pairs of integers [15], or bit-vectors by tuples of Booleans.

explain(P, e_0)	store_use(final_state$(P), e_0)$ $Ex(P)$
$Ex(s : I, P)$	$Ex(P)$ e_{post} = push_state(MEET(retrieve_uses(s))) $Ex_I(e_{post}, I)$
$Ex_I(e, \mathcal{T}^{\#}(\texttt{stmt})(q))$	store_use$(q,$ push_state$(E_{\mathcal{T}^{\#}}(\texttt{stmt})(e, q)))$
$Ex_I(e, \mathcal{T}^{-\#}(\texttt{stmt})(q))$	store_use$(q,$ push_state$(E_{\mathcal{T}^{-\#}}(\texttt{stmt})(e, q)))$
$Ex_I(e, \mathcal{R}^{\#}(\texttt{rel})(q))$	store_use$(q,$ push_state$(E_{\mathcal{T}^{-\#}}(\texttt{stmt})(e, q)))$
$Ex_I(e, \textsc{Join}([q_1, \ldots, q_n]))$	store_use(q_1, e); . . .; store_use(q_n, e)
$Ex_I(e, \textsc{Meet}([q_1, \ldots, q_n]))$	e' = push_state(E$\textsc{Meet}(e, [q_1, \ldots, q_n])$); store_use$(q_1, e'[1])$; . . .; store_use$(q_n, e'[n])$;
$Ex_I(e, \textsc{Rename}(q, [\texttt{x}_1, \ldots, \texttt{x}_n]))$	store_use$(q,$ push_state(E$\textsc{Rename}(e, q, [\texttt{x}_1, \ldots, \texttt{x}_n])$)
$Ex_I(e, \textsc{Splice}(q, q', [\texttt{x}_1, \ldots, \texttt{x}_n]))$	e' = push_state(E$\textsc{Splice}(e, q, q', [\texttt{x}_1, \ldots, \texttt{x}_n])$) store_use$(q, e'[1])$; store_use$(q', e'[2])$

Figure 7 Constructing an explanation from a propagator. The algorithm walks backwards along the computation, computing a sufficient postcondition for each instruction.

Just as we constructed a propagator from a checker in Section 4, we now define a corresponding translation scheme from propagators to explainers. It is assumed that the explanation algorithm is executed after the propagator, and has access to all the intermediate stages of the propagator. The primitive operations performed during explanation are $E_{\mathcal{T}^{\#}}(\texttt{stmt})(e, q)$, E$\textsc{Meet}(e, [q_1, \ldots, q_n])$, $E_{\mathcal{T}^{-\#}}(\texttt{stmt})(e, q)$, E$\textsc{Rename}(e, q, [x_1, \ldots, x_n])$, $E_{\mathcal{R}^{\#}}(\texttt{rel})(e, q)$, E$\textsc{Splice}(e, q, q', [x_1, \ldots, x_n])$, and $\textsc{Meet}([e_1, \ldots, e_n])$. $\textsc{Meet}$, as in the propagator case, simply conjoins a set of preconditions. All other operations simply push some postcondition back to the instruction's predecessor states (essentially computing an *interpolant* [10]).

The algorithm for translating a propagator into a corresponding 'explainer' is given in Figure 7. The explanation procedure runs backwards along the computations performed by the propagator, constructing a sufficient postcondition for each state of propagator state. In the propagator a given state may be used by multiple successors, particularly states corresponding to predicate heads and call sites. During explanation, each use of that state may result in a different postcondition. We use store_use to record the individual postconditions and conjoin them (using $\textsc{Meet}$) to construct an overall postcondition for the state before extrapolating back to the state's predecessors.

We have several choices in how this abstract explanation algorithm is embodied and used. A single run of the propagator may change domains of several variables. We may either generate a separate explanation for each domain change (which requires running the explanation algorithm several times), or construct a common explanation for all changes (which is cheaper, but yields less general explanations).

Another choice is how to represent preconditions. The most precise approach is to follow the same pattern as for propagation – maintain a full propagator state as the precondition and require externally provided *explanation transformers* for the necessary operations ($E_{\mathcal{T}^{\#}}$ and $E_{\mathcal{T}^{-\#}}$ for functions, $E_{\mathcal{R}^{\#}}$ for guards, and $E_{\sqcap}$ for meet), which turn a postcondition and incoming domains into a set of preconditions. Designing correct, efficient and precise implementations of explanation transformers is challenging, complicated by the fact that we need to deal with variables which are unconstrained in the postcondition (by either having an explicit $\top$ value, a Boolean flag, or pre-computing initial domains for each propagator state).

Domain representation: `t`, **Variable:** `v`, **Atomic constraint:** `a`			
Domain operations		Transformers	
equality	$(\mathtt{t},\mathtt{t}) \to \mathtt{bool}$	$\mathcal{T}^{\#}(\mathtt{fun})$	$\mathtt{list}(\mathtt{t}) \to \mathtt{t}$
conjunction	$(\mathtt{t},\mathtt{t}) \to (\mathtt{bool},\mathtt{t})$	$\mathcal{T}^{-\#}(\mathtt{fun})$	$(\mathtt{t},\mathtt{list}(\mathtt{t})) \to (\mathtt{bool},\mathtt{list}(\mathtt{t}))$
disjunction	$(\mathtt{t},\mathtt{t}) \to \mathtt{t}$	$\mathcal{T}^{\#}(\mathtt{rel})$	$(\mathtt{t},\mathtt{t}) \to (\mathtt{bool},(\mathtt{t},\mathtt{t}))$
Variable hooks		Explanation hooks	
GET-DOMAIN	$\mathtt{v} \to \mathtt{d}$	TO-ATOMS	$(\mathtt{v},\mathtt{d}) \to \mathtt{list}(\mathtt{a})$
SET-DOMAIN	$(\mathtt{v},\mathtt{d}) \to \mathtt{bool}$	SET-DOMAIN$_{expl}$	$(\mathtt{v},\mathtt{d},\mathtt{list}(\mathtt{a})) \to \mathtt{bool}$
		SET-CONFLICT	$\mathtt{list}(\mathtt{a}) \to \mathtt{unit}$

Figure 8 Operations that must be provided for domains, functions and relations in order to execute propagators.

We can instead construct a data-flow based explanation procedure. We track which values could have contributed to the inference of interest, and translate the corresponding domains to atomic constraints as the explanation. For the flow-based explanation, we represent the pre/post-condition as a pair (e_c, ν), where e_c indicates whether we must explain failure, and ν is a mapping from names to Booleans indicating whether the corresponding variable is relevant to the inference. Transformers for this analysis are straightforward. For example, $E_{\mathcal{T}^{\#}}(\mathtt{z\ =\ f(x,\ y)})((e_c,\nu),(c,\sigma))$ returns $(e_c, \nu[x \mapsto \nu(x) \vee \nu(z), y \mapsto \nu(y) \vee \nu(z)] \setminus \{z\})$.

6 Filling in the gaps

For the propagator construction of Section 4, we are missing implementations of three critical elements: the lattice of domain abstractions, state transformers for function and relation symbols, and hooks to communicate with the solver.

The operations needed to implement propagators are given in Figure 8. These fall into two classes: operations on domain abstractions, and communication between the propagators and the underlying solver. A pleasant outcome of this separation is that domain operations are entirely decoupled from the underlying solver – once lattice operations and transformers are defined for a given domain, they may be re-used in other solvers. The only operations which must be defined per solver *and* per variable kind is the extraction and update of domains.

For a classical CP solver, these are the only operations which must be defined. For lazy clause generation, the solver must also provide operations for dealing with atomic constraints. From the propagators' perspective, atomic constraints are entirely opaque. The solver specifies the (maximum) atom size, and each variable indicates the maximum number of atoms required to explain its domain. Before setting domains, we allocate a buffer large enough to fit the largest possible explanation. TO-ATOMS writes atomic constraints to this buffer, returning the end of the explanation so far. SET-CONFLICT and SET-DOMAIN$_{expl}$ will then retrieve the explanation from this buffer.

7 Instantiating Constraints from Specifications

We now return to the problem of transforming high-level specifications into intermediate form. The process must make two transformations: eliminating nested arithmetic expressions, and unfolding predicate bodies. The first is done in the usual manner, introducing fresh variables for sub-terms.

The second amounts to partially evaluating the logic program under the given instantiation. When evaluating a predicate call $\mathtt{p}(T)$ (where $T = [\mathtt{t}_1, \ldots, \mathtt{t}_n]$), we compute 'canonical arguments' T' by replacing each variable appearing in T with the index of its first occurrence, and (recursively) instantiate a copy of `p` with these T'. The instantiation of `p` is a predicate taking one argument for each variable appearing in T'.

Pattern-matching in clause heads is statically resolved. Clauses with non-matching heads or type-mismatches in expressions (e.g. arithmetic expressions instantiated on compound terms) are discarded, as are those containing calls to a predicate with no feasible clauses. The requirement that recursive calls be *structurally decreasing* is so that we may be sure the instantiation process terminates.

▶ **Example 3.** Recall the specification of `lex_lt`, given in Figure 1. Consider instantiating the constraint $\mathtt{lex\_lt}([\mathtt{X}, \mathtt{Y}], [\mathtt{Z}, \mathtt{Z}])$. Numbering variables in order of occurrence, we obtain the canonical arguments $[[\mathtt{V}_1, \mathtt{V}_2], [\mathtt{V}_3, \mathtt{V}_3]]$. Instantiating the first clause body, we get $\mathtt{V}_1 < \mathtt{V}_3$. In the second clause body, we see a recursive call to `lex_lt`, with (instantiated) arguments $[[\mathtt{V}_2], [\mathtt{V}_3]]$.

Instantiating $\mathtt{lex\_lt}([\mathtt{V}_1], [\mathtt{V}_2])$, we again obtain $\mathtt{V}_1 < \mathtt{V}_2$ for the first clause. In the second clause, we reach a recursive call $\mathtt{lex\_lt}([], [])$. Both clauses of `lex_lt` fail due to pattern matching, which causes the second clause of $\mathtt{lex\_lt}([\mathtt{V}_1], [\mathtt{V}_2])$ to fail. This gives us the instantiated checker:

```
lex_lt3(V1,V2,V3) :- V1 < V3. lex_lt3(V1,V2,V3) :- V1 = V3, lex_lt4(V2,V3).
lex_lt4(V1,V2) :- V1 < V2.
```
◀

8 Experimental Evaluation

We have implemented a prototype library `creidhne`[2] implementing this method. The library provides a C++ interface, but is implemented in OCaml using the LLVM compiler framework [14] for code generation.

Two's complement arithmetic

Integer arithmetic in CP operates on a subset of $\mathbb{Z}$. In some applications, particularly model checking, we instead wish to reason under machine arithmetic – the *fixed-width* or *wrapped* integers, which are not typically supported by CP solvers. This domain has received some attention [1, 13], but is not a common inclusion in CP or LCG solvers.

We used `creidhne` to integrate (signed) wrapped integers into `chuffed`,[3] a lazy clause generation CP solver. No modifications were needed to the underlying solver engine. Wrapped integers variables were represented internally using existing integer variables, and existing atomic constraints re-used. Connecting `chuffed` with `creidhne` totalled 300 lines of C++, plus minor changes to the FlatZinc [2] parser to allow string literals in annotations. The lattice operations and state transformers were implemented as code emitters for LLVM, totalling around 350 lines of OCaml.

We tested the synthesized propagators on some error-localization problems using reified 8-bit machine arithmetic. The synthesized propagators appear competitive with the native decompositions. For programmed search, the absence of introduced variables helps noticeably. Note that 32-bit wrapped integers could not be implemented by decomposition.

[2] Available at `http://bitbucket.org/gkgange/creidhne`.
[3] `http://github.com/geoffchu/chuffed`

Table 1 Average time (in seconds) and backtracks on small error-localization problems, using programmed (seq) or activity-driven (act) search. # gives number of instances.

	#	`native`(seq)	`native`(act)	`creidhne`(seq)	`creidhne`(act)
SUMSQUARES	19	16.65 / 193800	0.51 / 4712	0.80 / 21190	0.41 / 4651
TRITYP	100	0.17 / 2707	0.05 / 384	0.10 / 286	0.61 / 42880

9 Related Work

The burden of formulating and implementing propagation algorithms is well recognised, and a number of intermediate languages and compilation approaches have been proposed, although none consider generating explanations.

The approach of [3] represents constraint checkers as finite-state automata augmented with a finite set of counters. A constraint is instantiated by decomposing the automaton into a conjunction of primitive constraints. In [4] constraints were formulated as predicates denoting Boolean formulae of primitive constraints. Inference rules were derived for Boolean operators to determine which (lazily instantiated) primitive constraints could potentially propagate during search.

In [16], the authors propose a propagator specification language for global constraints based on an extension of *indexicals* [19], and define a compiler backend for each supported solver. The indexical-based specifications allow more finer control of propagation, but the universe of types is fixed and propagation rules must be specified by hand.

The method of [12] directly shares our objective of inferring efficient imperative propagators from arbitrary constraints. This approach eagerly pre-computes the result of enforcing domain consistency for all values in the powerset of $\mathcal{D}(c)$, and compiles a lookup table from the results. This computes extremely efficient (and domain-consistent) propagators, but is feasible only for constraints with small domains – the pre-computation time and worst-case memory requirements are $O(|\mathcal{P}D|^{|vars(c)|})$. For global constraints over integer variables with large domains, this approach is impractical.

Several existing works have applied ideas from abstract interpretation to constraint programming. The observation of constraint propagation as a fixpoint procedure was used in [18] to design an abstract interpretation based constraint solver for real variables. In [11], techniques from abstract interpretation were used to support constraints involving loops in a CLP formalism. These constraints were propagated by computing an approximation of the loop under the polyhedra abstract domain, then projecting back onto the problem variables.

10 Conclusion and Further Work

We have presented a system for synthesizing propagators (with explanation) over novel variable types from declarative specifications, and illustrated its effectiveness. There are numerous potential extensions, both in terms of the specification and the synthesis. These include adding support for partial functions, exploiting opportunities for more efficient propagation, and relaxing the restriction on unbounded recursion.

References

1 Sébastien Bardin, Philippe Herrmann, and Florian Perroud. An Alternative to SAT-Based Approaches for Bit-Vectors. In *Tools and Algorithms for the Construction and Analysis of Systems*, number 6015 in LNCS, pages 84–98. Springer Berlin Heidelberg, March 2010.

2 Ralph Becket. Specification of FlatZinc. [Online, accessed 3 March 2015], 2012. `http://www.minizinc.org/downloads/doc-1.6/flatzinc-spec.pdf`.

3 N. Beldiceanu, M. Carlsson, and T. Petit. Deriving filtering algorithms from constraint checkers. In *CP 2014*, volume 3258, pages 107–122, 2004. `doi:10.1007/978-3-540-30201-8_11`.

4 Sebastian Brand and Roland H. C. Yap. Towards 'propagation = logic + control'. In *ICLP 2006*, volume 4079, pages 102–116, 2006. `doi:10.1007/11799573_10`.

5 Rafael Caballero, Peter J. Stuckey, and Antonio Tenorio-Fornes. Two type extensions for the constraint modelling language MiniZinc. *Science of Computer Programming*, 111:156–189, 2016.

6 Geoffrey Chu. *Improving Combinatorial Optimization.* PhD thesis, Department of Computing and Information Systems, University of Melbourne, 2011.

7 Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL'77*, pages 238–252, New York, NY, USA, 1977. `doi:10.1145/512950.512973`.

8 Patrick Cousot and Radhia Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In *PLILP'92*, volume 631, pages 269–295, 1992.

9 Patrick Cousot, Radhia Cousot, and Francesco Logozzo. Precondition Inference from Intermittent Assertions and Application to Contracts on Collections. In *VMCAI 2011*, number 6538 in LNCS, pages 150–168. Springer Berlin Heidelberg, January 2011.

10 William Craig. Three uses of the herbrand-gentzen theorem in relating model theory and proof theory. *Journal of Symbolic Logic*, 22:269–285, 9 1957. `doi:10.2307/2963594`.

11 Tristan Denmat, Arnaud Gotlieb, and Mireille Ducassé. An abstract interpretation based combinator for modelling while loops in constraint programming. In *CP 2013*, volume 4741, pages 241–255, 2007. `doi:10.1007/978-3-540-74970-7_19`.

12 Ian P. Gent, Christopher Jefferson, Steve Linton, Ian Miguel, and Peter Nightingale. Generating custom propagators for arbitrary constraints. *Artif. Intell.*, 211:1–33, 2014. `doi:10.1016/j.artint.2014.03.001`.

13 Arnaud Gotlieb, Michel Leconte, and Bruno Marre. Constraint solving on modular integers. In *ModRef Workshop, associated to CP'2010*, September 2010.

14 C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis transformation. In *CGO 2004*, pages 75–86, March 2004. `doi:10.1109/CGO.2004.1281665`.

15 Christopher Mears, Andreas Schutt, Peter J. Stuckey, Guido Tack, Kim Marriott, and Mark Wallace. Modelling with option types in minizinc. In *CPAIOR 2014*, number 8451 in LNCS, pages 88–103. Springer, 2014. `doi:10.1007/978-3-319-07046-9_7`.

16 Jean-Noël Monette, Pierre Flener, and Justin Pearson. Towards solver-independent propagators. In *CP 2012*, volume 7514, pages 544–560, 2012. `doi:10.1007/978-3-642-33558-7_40`.

17 O. Ohrimenko, P.J. Stuckey, and M. Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009.

18 Marie Pelleau, Antoine Miné, Charlotte Truchet, and Frédéric Benhamou. A Constraint Solver Based on Abstract Domains. In *VMCAI 2013*, number 7737 in LNCS, pages 434–454. Springer Berlin Heidelberg, January 2013.

19 P. Van Hentenryck, Vijay Saraswat, and Yves Deville. Constraint processing in cc(FD). Technical report, Computer Science Department, Brown University, 1992.