Polyvariant Program Specialisation with Property-based Abstraction

John P. Gallagher
Roskilde University, Denmark and IMDEA Software Institute, Madrid, Spain
jpg@ruc.dk

In this paper we show that property-based abstraction, an established technique originating in software model checking, is a flexible method of controlling polyvariance in program specialisation in a standard online specialisation algorithm. Specialisation is a program transformation that transforms a program with respect to given constraints that restrict its behaviour. Polyvariant specialisation refers to the generation of two or more specialised versions of the same program code. The same program point can be reached more than once during a computation, with different constraints applying in each case, and polyvariant specialisation allows different specialisations to be realised. A property-based abstraction uses a finite set of properties to define a finite set of abstract versions of predicates, ensuring that only a finite number of specialised versions is generated. The particular choice of properties is critical for polyvariance; too few versions can result in insufficient specialisation, while too many can result in an increase of code size with no corresponding efficiency gains. Using examples, we show the flexibility of specialisation with property-based abstraction and discuss its application in control flow refinement, verification, termination analysis and dimension-based specialisation.

1 Program specialisation

Specialisation is a program transformation that transforms a program with respect to some given constraints that restrict its behaviour. A classic example is the loop in Figure 1(a) for computing $\mathtt{z} = \mathtt{x}^{\mathtt{y}}$. Figure 1(b) shows the result of specialising the loop with the input constraint $\mathtt{y} = 3$, unfolding the loop three times and evaluating the statement `y--` in the loop body.

```
z = 1;
while (y>0) {
  z = x*z; y--;
}
```

(a)

```
/* Input constraint y=3 */
z = 1; z = x*z; z = x*z; z = x*z;
```

(b)

Figure 1: Specialisation of a loop

Some specialisation methods could further transform the code in Figure 1(b) to `z = x*x*x;` using algebraic reasoning. As well as exploiting input constraints to specialise the program, we can perform internal specialisations based on constraints generated during program execution. For example, when specialising a statement $\mathtt{if}(e)\{s_1\}\{s_2\}$, even where the test e itself cannot be evaluated, the branch s_1 can be specialised with the constraint e and the branch s_2 can be specialised with the constraint $\neg e$. This is sometimes called *driving* [27].

Polyvariant specialisation refers to the generation of two or more specialised *versions* of the same program code. For example suppose that the statement $\mathtt{if}(x < 100)\{s_1\}\{s_2\}$ is reached twice during a

A.P. Lisitsa and A.P. Nemytykh (Eds.): VPT 2019
EPTCS 299, 2019, pp. 34–48, doi:10.4204/EPTCS.299.6

© J. P. Gallagher
This work is licensed under the
Creative Commons Attribution License.

```
while (x>0) {                 if (x>0) {
  if (y<m) {                    while (y<m) { /* x>0 */
    y++;                          y++;}
  } else {                      x--;
    x--;                        while (x>0) { /* y>=m */
  }                               x--;}
}                             }
```

(a) (b)

Figure 2: Polyvariant specialisation of a loop

computation, once with the constraint $x < 100$ and the other with the constraint $x \geq 100$. Polyvariant specialisation gives rise to two instances of the statement in the specialised code, s_1 and s_2 respectively.

Figure 2 illustrates both internal specialisation and polyvariance. The "then" branch of the **if** statement in Figure 2(a) does not affect the loop condition and so if it is taken, the "then" branch is repeatedly taken until the test $\mathtt{y} < \mathtt{m}$ fails. Then the "else" statement is executed (`x--;`) after which the "else" branch is repeatedly taken, since it does not affect the condition $\mathtt{y} < \mathtt{m}$, until the test $\mathtt{x} > \mathtt{0}$ fails.

Thus implicitly there are two distinct loops separated by `x--;` and this leads to the polyvariant specialisation shown in Figure 2(b). The loops in Figure 2(b) are reconstructed from an internal control flow representation; the first **while** loop has a loop test corresponding to the **if** statement from the input program. Further explanation of this example is given in Example 3.

The main contribution of this paper is a specialisation algorithm that performs polyvariant specialisation. Instances of this algorithm have been previously used and briefly described [18, 19, 9] but these papers did not present and discuss the general algorithm. A key question is the control of polyvariance; in general there could be many (even an infinite number) of possible variants of a given program point. How does the specialisation algorithm determine a suitable set of variants, while ensuring termination of the specialisation algorithm.

The algorithm operates on constrained Horn clauses, which provide a representation language capable of representing the semantics of a wide range of programming languages and systems. The algorithm is parameterised by a set of properties that control the generation of polyvariance.

2 Preliminaries

2.1 Constraints and entailment

Let T be a theory and let $\mathscr{C}^T$ be the set of formulas (also called *constraints*) constructed from the predicates and function symbols of T together with variables and boolean connectives, and the formulas true and false. $\models_T \phi$ means that ϕ is true in T, where ϕ is a variable-free formula. For example, $\models_T 1 \geq 0$ where T is the theory of linear real arithmetic (LRA). In this paper, if we omit the theory subscript T we assume that T is the theory of linear real arithmetic, and we omit the symbol $\models$ when clear from context.

Let $\phi \in \mathscr{C}^T$ be a constraint possibly containing variables; a substitution for the variables of ϕ is a *grounding substitution* if the result of applying the substitution to ϕ, say ϕ', contains no variables; if $\models_T \phi'$ the grounding substitution *satisfies* ϕ.

For all constraints ϕ and ψ, we say that ϕ *entails* ψ in T, written $\phi \preceq_T \psi$, if every grounding substitution that satisfies ϕ in T also satisfies ψ in T. For example, $x \geq 1 \preceq_T x \geq 0$ where T is LRA.

We assume that there is a procedure called SAT_T such that for every $\phi \in \mathscr{C}^T$, $\mathsf{SAT}_T(\phi)$ is true if there is some substitution that satisfies ϕ and false otherwise. Using SAT_T, we can check entailment; $\phi \preceq_T \psi$ if and only if $\mathsf{SAT}(\phi \wedge \neg\psi)$ is false.

Definition 1 (Generalisation) *A function $\rho : \mathscr{C}^T \rightarrow \mathscr{C}^T$ is called a* generalisation operator *if $\phi \preceq_T \rho(\phi)$.*

2.2 Constrained Horn clause representation of programs

A constrained Horn clause (CHC) over some constraint theory T is a first-order predicate logic formula of the form $\forall \mathbf{x_0} \ldots \mathbf{x_k}(p_1(\mathbf{x_1}) \wedge \ldots \wedge p_k(\mathbf{x_k}) \wedge \phi \rightarrow p_0(\mathbf{x_0}))$, where ϕ is a finite conjunction of *constraints* from $\mathscr{C}^T$, $\mathbf{x_0}, \ldots, \mathbf{x_k}$ are (possibly empty) tuples of *variables*, $p_0, \ldots, p_k$ are *predicate symbols*, $p_0(\mathbf{x_0})$ is the *head* of the CHC and $p_1(\mathbf{x_1}) \wedge \ldots \wedge p_k(\mathbf{x_k}) \wedge \phi$ is the *body*. Formulas of the form $p(\mathbf{x})$ are called atomic formulas or simply *atoms*. A CHC is often written as $p_0(\mathbf{x_0}) \leftarrow \phi, p_1(\mathbf{x_1}), \ldots, p_k(\mathbf{x_k})$ in the style of constraint logic programs, or `p0(X0) ← C, p1(X1),....,pk(Xk)` in text form, where `C` is a constraint formula. A *constrained fact* is a CHC $p(\mathbf{x}) \leftarrow \phi$, where ϕ is a constraint over $\mathbf{x}$.

Definition 2 (Ordering on sets of constrained facts) *We extend the relation $\preceq$ to sets of constrained facts. Let S, S' be sets of constrained facts. Then $S \preceq S'$ if for each constrained fact $p(\mathbf{x}) \leftarrow \phi$ in S there exists a constrained fact (with variables suitably renamed) $p(\mathbf{x}) \leftarrow \psi$ in S', such that $\phi \preceq \psi$. Furthermore, if ρ is a generalisation operator on constraints then $S \preceq \{p(\mathbf{x}) \leftarrow \rho(\phi) \mid p(\mathbf{x}) \leftarrow \phi \in S\}$.*

We do not go into detail on the translation of imperative programs to CHCs, but note that a distinct predicate symbol is generated for each program point, and the arguments of the predicate for a given program point are the values of the program variables at that point. The CHCs defining a predicate capture the transitions in an operational semantics. Figure 3 illustrates the translation of the programs in Figures 1(a) and 2(a) into CHCs. Depending on the style of semantic specification (such as small-step semantics or big-step semantics), different CHCs can be obtained for a program.

```
start ←
  p0(X,Y,Z).
p0(X,Y,Z) ←
  Z1=1,
  while0(X,Y,Z1).
while0(X,Y,Z) ←
  Y>0, Z1=X*Z, Y1=Y-1,
  while0(X,Y1,Z1).
while0(X,Y,Z) ←
  Y=<0.
```

(a)

```
start ←
  while0(X,Y,M).
while0(X,Y,M) ←
  X>0,
  if0(X,Y,M).
while0(X,Y,M) ←
  X=<0.
if0(X,Y,M) ←
  Y<M, Y1=Y+1,
  while0(X,Y1,M).
if0(X,Y,M) ←
  Y>=M, X1=X-1,
  while0(X1,Y,M).
```

(b)

Figure 3: CHC representation of (a) Figure 1(a) and (b) Figure 2(a)

2.3 Constrained Horn clause derivations

The definitions of CHC derivations and partial evaluation are based on standard definitions (e.g. [22]) adapted to include constraints, and using the "resultant" style of derivation. Instead of an initial query or goal $\leftarrow \phi, p(\mathbf{x})$, we start a derivation with a CHC $p(\mathbf{x}) \leftarrow \phi, p(\mathbf{x})$ and each step replaces a body literal.

Definition 3 (Derivation step) *A CHC* derivation step *or* unfolding step *is defined as follows. Let c_1, c_2 be CHCs, where $c_1 = q_0(\mathbf{x_0}) \leftarrow \phi, q_1(\mathbf{x_1}), \ldots, q_k(\mathbf{x_k})$ and $c_2 = q_i(\mathbf{y_0}) \leftarrow \phi', r_1(\mathbf{y_1}), \ldots, r_m(\mathbf{y_m})$, with variables of c_1 and c_2 renamed apart. Then the result of* unfolding c_1 with c_2 on $q_i(\mathbf{x_i})$ *is:*

$$\begin{cases} q_0(\mathbf{x_0}) \leftarrow \phi \wedge \phi' \wedge \mathbf{x_i} = \mathbf{y_0}, \\ \quad q_1(\mathbf{x_1}), \ldots, q_{i-1}(\mathbf{x_{i-1}}), \\ \quad r_1(\mathbf{y_1}), \ldots, r_m(\mathbf{y_m}), \\ \quad q_{i+1}(\mathbf{x_{i+1}}), \ldots, q_k(\mathbf{x_k}) & \textit{if } \mathsf{SAT}(\phi \wedge \phi' \wedge \mathbf{x_i} = \mathbf{y_0}) \\ q_0(\mathbf{x_0}) \leftarrow \mathsf{false} & \textit{otherwise} \end{cases}$$

Definition 4 (Derivation tree) *Let P be a set of CHCs and let $A = p(\mathbf{x}) \leftarrow \phi$ be a constrained fact where ϕ is a constraint on $\mathbf{x}$. Then a* derivation tree *for A in P is a tree where every node is labelled by a CHC, such that:*

- *the root is labelled with $p(\mathbf{x}) \leftarrow \phi, p(\mathbf{x})$;*
- *for a non-leaf node labelled with a CHC c, its children are labelled with CHCs $\{c_1, \ldots, c_k\}$, where $q_i(\mathbf{x_i})$ is some atom in the body of c, $\{d_1, \ldots, d_k\}$ is the set of CHCs (with variables suitably renamed) in P, whose head has predicate q_i, and c_i is the result of unfolding c with d_i on $q_i(\mathbf{x_i})$;*
- *a leaf node is labelled by a CHC of the form $p(\mathbf{x}) \leftarrow \varphi$, where φ is a constraint (possibly* false*).*

The definition is non-deterministic since any atom $q_i(\mathbf{x_i})$ can be selected at a non-root node.

A derivation tree can contain infinite branches. The specialisation algorithm developed in the next section depends on constructing a *partial* derivation tree, which is a finite tree following Definition 4 with the additional case that a leaf node may be labelled by a CHC of the form $p(\mathbf{x_0}) \leftarrow \phi, q_1(\mathbf{x_1}), \ldots, q_k(\mathbf{x_k})$ ($k > 0$) representing an incomplete branch of the derivation tree. A branch of a (partial) derivation tree is a *failing* branch if it ends in a leaf labelled by $p(\mathbf{x}) \leftarrow$ false, otherwise it is *non-failing*.

Definition 5 (Partial evaluation of a constrained fact) *Let A be a constrained fact and P be a set of CHCs. Let T be a derivation tree for A in P. Then a* partial evaluation of *A* in *P is a finite set of CHCs $\{c_1, \ldots, c_m\}$ labelling nodes chosen from the non-root nodes of T such that there is exactly one node for each non-failing branch of T.*

Clearly, the whole derivation tree does not have to be constructed in order to get a partial evaluation, but only an initial portion; then one CHC from each branch is collected.

The non-determinism in the definition of a derivation tree is resolved by an *unfolding rule*, which both selects which body atom to unfold at each step, and decides when to stop extending a branch in a (partial) derivation tree in order to return a partial evaluation.

Definition 6 (Unfolding rule) *An unfolding rule U is a function which given a set of CHCs P and a constrained fact A, returns exactly one finite set of CHCs that is a partial evaluation of A in P. For a set S of constrained facts, the set of CHCs obtained by applying U to each element of S is called a partial evaluation of S in P using U.*

Example 1 *Let P be the set of clauses in Figure 3(b). Let A be the constrained atom* `while0(X,Y,M)` $\leftarrow$ `true`. *Figure 4 shows three sets of CHCs that are examples of partial evaluations of A in P, with different unfolding rules. Note that for CHCs with at most one atom in the body, such as in this example, the choice of atom for each derivation step is determinate, so the unfolding rule only determines how far to unfold each branch.*

```
while0(X,Y,M) ←
   X>0,
   if0(X,Y,M).
while0(X,Y,M) ←
   X=<0.
```

```
while0(X,Y,M) ←
   X>0,
   Y<M, Y1=Y+1,
   while0(X,Y1,M).
while0(X,Y,M) ←
   X>0,
   Y>=M, X1=X-1,
   while0(X1,Y,M).
while0(X,Y,M) ←
   X=<0.
```

```
while0(X,Y,M) ←
   X>0,
   Y<M, Y1=Y+1,
   X>0,
   if0(X,Y1,M).
while0(X,Y,M) ←
   X>0,
   Y<M, Y1=Y+1,
   X=<0.
while0(X,Y,M) ←
   X>0,
   Y>=M, X1=X-1,
   while0(X1,Y,M).
while0(X,Y,M) ←
   X=<0.
```

Figure 4: Three possible partial evaluations of constrained fact `while0(X,Y,M)` $\leftarrow$ `true` in Figure 3(b). The leftmost column is the trivial unfolding, consisting of the original clauses for `while0`.

3 Specialisation algorithm

Algorithm 1 shows an outline specialisation algorithm SP for CHCs based on the “basic algorithm” in [12]. This is a so-called *online* specialisation algorithm, which makes control decisions on evaluation and polyvariance on the fly, as opposed to *offline* specialisation, in which control decisions are determined by the results of a prior analysis such as a binding time analysis.

The algorithm takes as input a set of CHCs P and a set of entry points S_0, where each entry point is a constrained fact. It is parameterised by two operations, namely pe and α_ρ.

- $\mathsf{pe}(S)$ returns a partial evaluation (Definition 5) of the set of constrained facts S.
- $\alpha_\rho(S)$ is a set of constrained facts such that $\alpha_\rho(S) = \{p(\mathbf{x}) \leftarrow \rho(\phi) \mid p(\mathbf{x}) \leftarrow \phi \in S\}$, where ρ is some generalisation operator (Definition 1).

Two other functions are called in the algorithm, $\mathsf{collect}$ and $\mathsf{unfoldfold}$.

- $\mathsf{collect}(Q)$ returns the set of constrained atoms collected from the bodies of clauses in a set of CHCs Q. It is defined as $\mathsf{collect}(Q) = \{p_i(\mathbf{x_i}) \leftarrow \phi|_{\mathbf{x_i}} \mid p_0(\mathbf{x_0}) \leftarrow \phi, p_1(\mathbf{x_1}), \ldots, p_k(\mathbf{x_k}) \in Q\}$.

The $\mathsf{unfoldfold}$ function is an unfold-fold transformation, which will be discussed in Section 3.1 and in Example 3.

Algorithm 1 SP(P,S_0)

1: **Input:** Finite set of CHCs P, finite set of constrained facts S_0, generalisation operator ρ.
2: **Output:** Finite set of CHCs
3: $S \leftarrow S_0$
4: **repeat**
5:     $S' = S$
6:     $S \leftarrow S \cup \alpha_\rho(\mathsf{collect}(\mathsf{pe}(S)))$
7: **until** $S' = S$
8: **return** $\mathsf{unfoldfold}(S)$

The successive values of S' in the **repeat** loop of the algorithm (line 7) form an increasing sequence

of sets of constrained facts with respect to $\preceq$ starting from the input set S_0, say $S_0, S_1, S_2, \ldots$; the loop terminates if for some $j > 0$, $S_{j-1} = S_j$.

3.1 Correctness of the specialisation algorithm

The loop termination condition on line 7 of Algorithm 1 establishes a *closedness* condition.

Lemma 1 *Let S be the final set of constrained facts when the loop terminates. Then* $\mathsf{collect}(\mathsf{pe}(S)) \preceq S$.

The proof of the lemma relies directly on the fact that α_ρ in line 6 uses a generalisation operator (Definition 1). Lemma 1 is the basis for a proof of correctness using unfold-fold proofs [24, 10, 8]. Line 8 of the algorithm is defined as an *unfold-fold proof*, where folding is performed using a set of new definitions constructed from the set S obtained from the loop. Specifically, $\mathsf{Def}_S = \{p'(\mathbf{x}) \leftarrow \phi, p(\mathbf{x}) \mid p(\mathbf{x}) \leftarrow \phi \in S\}$, where p' is a fresh predicate symbol unique for each element of S. These definitions are unfolded, and then folded using Def_S. Lemma 1 guarantees that all the predicates from the input clauses can be folded to their renamed versions in Def_S. Thus in the final transformed program, every predicate in the head of a clause in Def_S calls only head predicates in Def_S. The clauses returned by $\mathsf{unfoldfold}(S)$are just those defining head predicates in Def_S, since the original predicates are unreachable from initial set of queries S_0. (In practice, we can then rename versions from S_0 back to their original predicate names.)

In short, the main loop of Algorithm 1 constructs a set of new definitions Def_S, while correctness of the clauses returned by the algorithm follows from the general results on unfold-fold transformations using Def_S, along with the closeness property of Lemma 1.

4 Property-based abstraction

We now turn to the consideration of the abstraction function α_ρ, using property-based abstraction. We first define a generalisation operator ρ_Ψ. Let $\Psi \subseteq \mathscr{C}^T$ be a finite set of constraints. Given a formula $\phi \in \mathscr{C}^T$, then

$$\rho_\Psi(\phi) = \bigwedge\{\psi \mid \psi \in \Psi, \phi \preceq \psi\} \wedge \bigwedge\{\neg\psi \mid \psi \in \Psi, \phi \preceq \neg\psi\}.$$

Lemma 2 ρ_Ψ *is a generalisation operator, that is, for all* $\phi \in \mathscr{C}^T$, $\phi \preceq_T \rho_\Psi(\phi)$.

Note that $\rho_\Psi(\phi)$ is a conjunction of elements of Ψ and negations of elements of Ψ. Since we assume that Ψ is finite, then the set of possible values of $\rho_\Psi(\phi)$ is finite. If none of the elements of Ψ or their negations is entailed by ϕ then $\rho_\Psi(\phi) = \mathsf{true}$.

Example 2 *Consider a set* $\Psi = \{\psi_1, \psi_2, \psi_3\}$, *where* $\psi_1 \wedge \psi_3 = \mathsf{false}$, $\psi_2 \wedge \psi_3 = \mathsf{false}$ *and* $\psi_1 \wedge \psi_2 \neq \mathsf{false}$. *Figure 5 shows the generalisation for various choices of a property* φ, *that is, the values of* $\rho_\Psi(\varphi)$.

ρ_Ψ is extended to apply to constrained facts, and for convenience we take the set Ψ also to consist of constrained facts.

$$\rho_\Psi(p(\mathbf{x}) \leftarrow \phi) = p(\mathbf{x}) \leftarrow \bigwedge\{\psi \mid p(\mathbf{x}) \leftarrow \psi \in \Psi, \phi \preceq \psi\} \wedge \bigwedge\{\neg\psi \mid p(\mathbf{x}) \leftarrow \psi \in \Psi, \phi \preceq \neg\psi\}.$$

Let S and Ψ be sets of constrained facts. The operation α_ρ from Algorithm 1 is defined where ρ is the generalisation operator ρ_Ψ.

$$\alpha_{\rho_\Psi}(S) = \{p(\mathbf{x}) \leftarrow \rho_\Psi(\varphi) \mid p(\mathbf{x}) \leftarrow \varphi \in S\}$$

We now have all the components of the algorithm, and we show an example of specialisation with property-based abstraction.

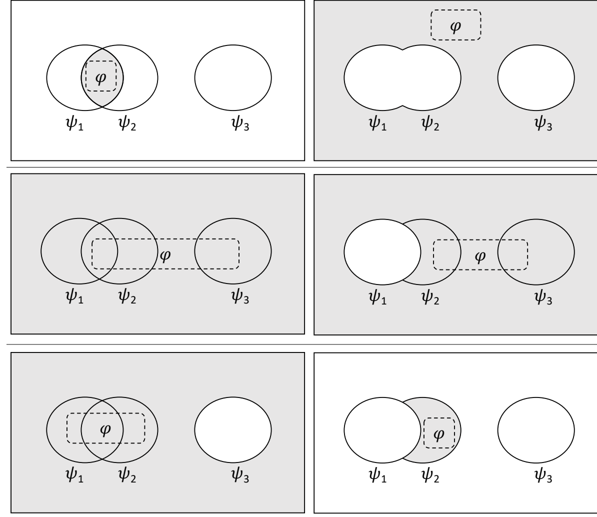

Figure 5: The property to be generalised (φ) is shown as an area with dotted outline, and the shaded areas show the generalisation of φ using operation $\rho_\Psi(\varphi)$, where $\Psi = \{\psi_1, \psi_2, \psi_3\}$.

Example 3 *Let P be the set of CHCs representing the code in Figure 2(a), that is, the clauses from Figure 3(b). Let $S_0 = \{\texttt{start} \leftarrow \texttt{true}\}$ and let the set Ψ contain the following constrained facts.*

```
while0(A,B,C) ← A>0      while0(A,B,C) ← B<C      if0(A,B,C) ← B<C
while0(A,B,C) ← A≤0      while0(A,B,C) ← B≥C      if0(A,B,C) ← B≥C
```

The following unfolding rule is used: a clause body is unfolded until either a branch point is reached (i.e. a call to a predicate that appears in the head of more than one clause) or a recursive predicate is reached (i.e. a predicate that is the target of a back edge in the predicate dependency graph of the program traversed from the initial predicate `start`). In the given clauses, this implies that each partial evaluation consists of only one unfolding step since every predicate is either recursive or a branch. Algorithm 1 proceeds as follows, with S_0 initialised to $\{\texttt{start} \leftarrow \texttt{true}\}$.

- *Iteration 1:* $S_1 = S_0 \cup \{\texttt{while0(A,B,C)} \leftarrow \texttt{true}\}$.
- *Iteration 2:* $S_2 = S_1 \cup \{\texttt{if0(A,B,C)} \leftarrow \texttt{A} > \texttt{0}\}$.
- *Iteration 3:* $S_3 = S_2 \cup \{\texttt{while0(A,B,C)} \leftarrow \texttt{A} > \texttt{0},\quad \texttt{while0(A,B,C)} \leftarrow \texttt{B} \geq \texttt{C}\}$.
- *Iteration 4:* $S_4 = S_3 \cup \{\texttt{if0(A,B,C)} \leftarrow \texttt{A} > \texttt{0}, \texttt{B} \geq \texttt{C}\}$.
- *Iteration 5:* $S_5 = S_4$.

In this sequence we applied the operator α_{ρ_Ψ} at each stage to reach the sets shown in the sequence. (To be precise, we apply the generalisation operator only to constrained facts for the recursive predicate `while0`*, which is sufficient to ensure termination). For example, in iteration 3, partial evaluation of the constrained fact $\texttt{if0(A,B,C)} \leftarrow \texttt{A} > \texttt{0}$ results in the following CHCs.*

```
if0(A,B,C) ← A>0,C>B,D=B-1,while0(A,D,C).
if0(A,B,C) ← A>0,B≥C,D=A-1,while0(D,B,C).
```

The constraints projected onto the body atoms $\texttt{while0(A,D,C)}$ *and* $\texttt{while0(D,B,C)}$ *are respectively* $\texttt{A} > \texttt{0}$ *and* $\texttt{D} > -\texttt{1}, \texttt{B} \geq \texttt{C}$*. The result of applying abstraction, that is,*

$$\alpha_{\rho_\Psi}(\{\texttt{while0(A,D,C)} \leftarrow \texttt{A} > \texttt{0},\quad \texttt{while0(D,B,C)} \leftarrow \texttt{D} > -\texttt{1}, \texttt{B} \geq \texttt{C}\})$$

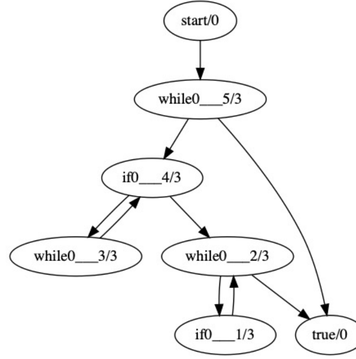

Figure 6: The predicate dependency graph for Example 3

is $\{\mathtt{while0(A,B,C)} \leftarrow \mathtt{A} > 0, \mathtt{while0(A,B,C)} \leftarrow \mathtt{B} \geq \mathtt{C}\}$. *Notice that the second constrained fact has been generalised; evaluating* $\rho_\Psi(\mathtt{while0(D,B,C)} \leftarrow \mathtt{D} > -1, \mathtt{B} \geq \mathtt{C}\})$ *yields* $\mathtt{while0(A,B,C)} \leftarrow \mathtt{B} \geq \mathtt{C}$*, since* $\mathtt{B} \geq \mathtt{C}$ *is the only element of* Ψ *that is entailed. Similarly, in iteration 5, partial evaluation of* $\mathtt{if0(A,B,C)} \leftarrow \mathtt{A} > 0, \mathtt{B} \geq \mathtt{C}$ *results in the CHC:*

```
if0(A,B,C) ← A>0,B≥C,D=A-1,while0(D,B,C).
```

The constraint on the body atom $\mathtt{while0(D,B,C)}$ *is* $\mathtt{D} > -1, \mathtt{B} \geq \mathtt{C}$*; as before this is generalised to the constrained fact* $\mathtt{while0(A,B,C)} \leftarrow \mathtt{B} \geq \mathtt{C}$.

The function $\mathsf{unfoldfold}_\rho$ *of Algorithm 1 constructs a set of renaming definitions, as follows.*

$$\begin{array}{l|l} \mathtt{start} \leftarrow \mathtt{start}, \mathtt{true} & \mathtt{if0_1(A,B,C)} \leftarrow \mathtt{if0(A,B,C)}, \mathtt{A} > 0, \mathtt{B} \geq \mathtt{C} \\ \mathtt{while0_2(A,B,C)} \leftarrow \mathtt{while0(A,B,C)}, \mathtt{B} \geq \mathtt{C} & \mathtt{while0_3(A,B,C)} \leftarrow \mathtt{while0(A,B,C)}, \mathtt{A} > 0 \\ \mathtt{if0_4(A,B,C)} \leftarrow \mathtt{if0(A,B,C)}, \mathtt{A} > 0 & \mathtt{while0_5(A,B,C)} \leftarrow \mathtt{while0(A,B,C)}, \mathtt{true} \end{array}$$

These definitions are unfolded (using the same strategy as for the pe *operation) and the atoms in the bodies of the unfolded clauses are folded using the above clauses. This gives the following specialised CHC clauses.*

```
start ← while0_5(A,B,C).
while0_5(A,B,C) ← A>0,if0_4(A,B,C).
while0_5(A,B,C) ← -A>=0.
if0_4(A,B,C) ← A>0,-B+C>0,B-D= -1,while0_3(A,D,C).
if0_4(A,B,C) ← A>0,B-C>=0,A-D=1,while0_2(D,B,C).
while0_3(A,B,C) ← A>0,if0_4(A,B,C).
while0_2(A,B,C) ← B-C>=0,A>0,if0_1(A,B,C).
while0_2(A,B,C) ← B-C>=0,-A>=0.
if0_1(A,B,C) ← A>0,B-C>=0,A-D=1,while0_2(D,B,C).
```

The predicate dependency graph for these clauses is shown in Figure 6 and it can be seen that this has the same structure as the code in Figure 2(b) where there are two distinct loops. Note that the predicate $\mathtt{while0_5}$ *is in fact not the head of a loop but rather the initial if-statement of Figure 2(b), while the predicate* $\mathtt{if0_4}$ *is the head of a loop.*

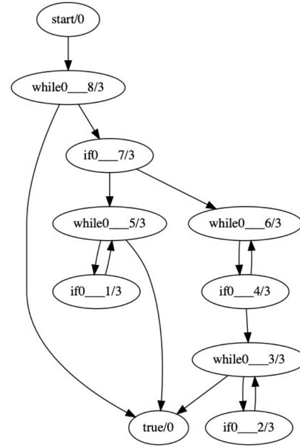

Figure 7: The predicate dependency graph for Example 3, with enlarged set of properties.

5 Choice of properties and granularity of abstraction

The set of properties Ψ in Example 3 was chosen so that the properties were relevant to the tests determining the control flow. However, the choice of properties can be critical to achieving good specialisations. In this section we discuss the effect of the choice of properties.

In general, it is clear that the larger the set of properties, the more versions of predicates can be produced, and thus more specialised clauses can be generated. Fewer properties, on the other hand, cause information needed for specialisation to be lost. For example, the following properties could also be chosen in Example 3. It is a subset of the set previously chosen, incorporating only the constraints directly appearing in the clauses for the respective predicates.

```
{ while0(A,B,C) ← A>0, while0(A,B,C) ← A≤0,
  if0(A,B,C) ← B<C, if0(A,B,C) ← B≥C }
```

Using this choice of Ψ, no specialisation at all is achieved; the original clauses are returned. The problem is that the constraints on `if0` are lost when abstracting the calls to `while0`, since the properties applying to `while0(A,B,C)` say nothing about the values of `B` or `C`.

However, for a given unfolding rule, there is a limit to how much specialisation can be achieved, no matter how many properties Ψ contains. Consider the following set Ψ for Example 3, which results from collecting all constraints from the given clauses, projected onto head and body atoms.

```
                                                     if0(A,B,C) ← A>0
while0(A,B,C) ← A>0     while0(A,B,C) ← C>B-1       if0(A,B,C) ← B<C
while0(A,B,C) ← A≤0     while0(A,B,C) ← B≥C         if0(A,B,C) ← B≥C
```

Figure 7 shows the predicate dependency graph for the clauses resulting from this set. While there are more versions of predicates than in Figure 6, the corresponding clauses are no more specialised. Viewing the graph as a finite automaton, it can be verified that the states `if0_4` and `if0_7` are equivalent, as are `while0_5` and `while0_3`, and `if0_1` and `if0_2`. In short, the automaton can be minimised to give the same automaton as in Figure 6. For this example, there can be no better specialisation for the input clauses in Example 3, with any set of properties Ψ or indeed any other unfolding rule, than the one achieved in Example 3. Space does not permit a detailed account of the automata-theoretic argument, but we state the following conjecture.

```
int a, b;
if (a ≤ 100)
   a = 100-a;
else
   a=a-100;
while (a ≥ 1)
   a=a-1;
   b=b-2;
assert(b != 0);
```

```
init(A,B) ← true.
if(A,B) ← A0 ≤ 100, A=100-A0, init(A0,B).
if(A,B) ← A0 ≥ 101, A=A0-100, init(A0,B).
while(A,B) ← if(A,B).
while(A,B) ← A0≥1, A=A0-1, B=B0-2,
             while(A0,B0).
false ← A≤0, B=0,
             while(A,B)
```

Figure 8: Example from [18]: (left) original program, (right) translation to CHCs

Proposition 1 *Let P be a set of CHCs and S_0 a set of constrained facts, Ψ a set of properties and U an unfolding rule. Let P' be the output of Algorithm 1 with P and S_0 as inputs, using Ψ and U. Let $\{p_1,\ldots,p_k\}$ be the set of predicates from P of which the predicates of P' are variants. Then there exists a set of clauses P'' having a minimal number of variants of the same set $\{p_1,\ldots,p_k\}$, that is equivalent to P' wrt to derivations starting with S_0.*

Proof sketch: P'' can be constructed by minimisation of a tree automaton derived from P'. Note that $\{p_1,\ldots,p_k\}$ could be a strict subset of the predicates of P and so P itself does not in general satisfy the condition for P''. Although P'' is minimal in the sense given above, further specialisation might be achievable using a different unfolding rule than U. We also conjecture that there exists a set of properties Ψ' such that executing Algorithm 1 with Ψ', U, P and S_0 would yield P''.

6 Polyvariant specialisation in verification

In this section we discuss the role of polyvariant specialisation of CHCs in program verification tasks.

Pre-condition inference. In [18], Algorithm 1 was used as a component in an algorithm for computing sufficient conditions for safety of imperative programs encoded as CHCs. In many cases, the safety condition is a disjunction. Polyvariant specialisation enabled the relevant disjuncts to be found by a convex polyhedral analysis.

Example 4 *Consider the example in Figure 8 taken from [18]. Note that the translation to CHCs corresponds to a backwards flow of control from the error predicate* `false` *to the program start predicate* `init`*. The goal is to infer conditions on the start predicate that ensure that the error predicate is not reached. Specialisation of the set of CHCs was carried out using the following set of properties, and the same unfolding rule as in Example 3.*

```
{if(A,B) ← A≥0, if(A,B) ← A≥1,
init(A,B) ← A≥101, init(A,B) ← -A≥ -100,
while(A,B) ← A≥0, while(A,B) ← A≥1, while(A,B) ← -A≥0,B=0,
while(A,B) ← -A≥0, while(A,B) ← B=0}
```

It can be seen that three versions of the predicate `init` *have been generated, arising from different paths through the program. Analysis of these specialised CHCs allowed the disjunctive precondition on* `init(A,B)`*, namely* $\mathtt{B} \neq |\mathtt{2A}-\mathtt{100}|$ *to be derived. This condition could not be derived from the original code without an analysis domain of disjunctive properties, which is more difficult to implement and control. Polyvariant specialisation, in effect, provides a heuristic for introducing disjunctions selectively, where they can affect the control flow.*

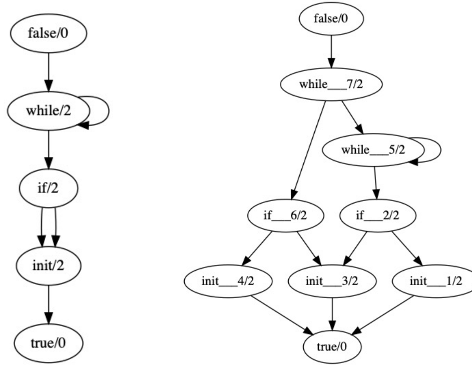

Figure 9: The predicate dependency graph for Example 4, before and after polyvariant specialisation

Termination analysis. In [9], polyvariant specialisation was used to transform a control-flow graph obtained from a program into another equivalent control-flow graph in which the loops were in a form more suitable for automatic proof of termination. The control-flow graphs are straightforwardly represented as CHCs and examples of their polyvariant specialisation are displayed in [9]. The example from Figure 2 provides a case in point. The structure of the single loop makes it rather hard to find a suitable ranking function that establishes termination; whereas the restructured code based on polyvariant specialisation, with two separate loops, is easy to prove terminating, since each loop has a simple ranking function. A large number of experiments was reported in [9], showing that polyvariant specialisation of the control-flow graph very often improves the effectiveness of both automatic termination analysis and complexity bound analysis. In short, the work demonstrates that control-flow refinement [16] can be achieved by polyvariant specialisation. Another relevant approach for finding better loop invariants is the "splitter predicates" method [26]; this can also be reproduced using property-based specialisation.

Dimension-based decomposition. The concept of *tree dimension* has been applied in verification to decompose a proof. The dimension of a set of CHCs is a measure of their non-linearity. A set of CHCs of dimension zero contains only linear clauses (that is, clauses having at most one atom in the body). Proof trees in such sets of CHCs have no branching. Sets of clauses of higher dimension give rise to branching proof trees, and the dimension of a tree is determined by the dimensions of the subtrees of the root, as illustrated in Figure 10. A more detailed definition can be found in [19]. Given a set of CHCs P for which some property is to be verified, we may decompose the problem by dimensions. For a given dimension d, we can define a set of CHCs, say P^d, such that an atom A has a proof in P of dimension d if and only if it has a proof in P^d. Since every proof has some finite dimension, $P \vdash A$ if and only if $P^0 \vdash A \vee P^1 \vdash A \vee P^2 \vdash A \ldots$. In [19], we showed a technique using polyvariant specialisation for generating various dimension-bounded sets of clauses. We could generate P^d, the set of clauses yielding exactly the proof trees of dimension d; we could also generate $P^{\leq d}$, the set of clauses yielding proof trees of dimension at most d; and we could generate $P^{>d}$, the set of clauses yielding proof trees of dimension at least $d+1$. This enabled a variety of strategies for proof decomposition, described in detail in [19].

In order to derive such dimension-constrained sets of clauses, we first instrumented the clauses with an extra argument for each predicate, representing the dimension, together with constraints capturing the rule for computing dimension. That is, the clause $p_0(\mathbf{x_0}) \leftarrow \phi, p_1(\mathbf{x_1}), \ldots, p_n(\mathbf{x_n})$ is replaced by

$$p_0(\mathbf{x_0}, k) \leftarrow \phi, p_1(\mathbf{x_1}, k_1), \ldots, p_n(\mathbf{x_n}, k_n), dim(k_1, \ldots, k_n, k)$$

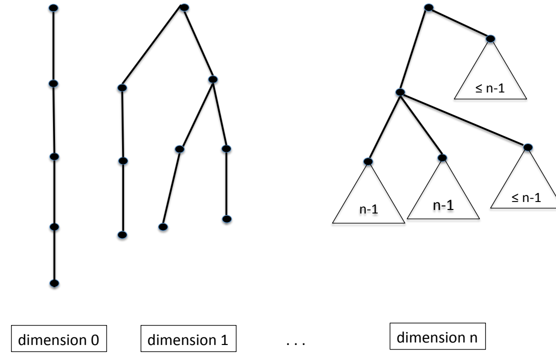

Figure 10: The dimension of a node in a tree: leaf nodes have dimension 0; a node has dimension $n+1$ if at least two subtrees have dimension n; it has dimension n if exactly one subtree has dimension n and any other subtrees have lower dimension.

where $dim(k_1,\ldots,k_n,k)$ represents the computation of the head dimension k from the (subtree) dimensions $k_1,\ldots,k_n$.

Specialisation was then performed with respect to constraints on k. The set of properties Ψ input to Algorithm 1 consisted a constrained fact for each dimension up to the required bound.

Example 5 *Let P be the set of clauses for the Fibonacci function, instrumented with the dimension as described above, together with a constraint representing a property to be proved and a dimension bound on* `false` *of 2.*

```
fib(A,B,0) :- A>=0, A=<1, A=B.
fib(A,B,K) :- A>1, D=A-2, E=A-1, B=F+G, fib(D,G,K2), fib(E,F,K1),
       K1+1=<K, K2=K.
fib(A,B,K) :- A>1, D=A-2, E=A-1, B=F+G, fib(D,G,K1), fib(E,F,K2),
       K1+1=<K, K=K2.
fib(A,B,K) :- A>1, D=A-2, E=A-1, B=F+G, fib(D,G,K1), fib(E,F,K2),
       K1=K-1, K2=K1.
false(A) ← X>5, fib(X,Y,K), Y<X, K≤2.
```

Let Ψ be the following set of constrained facts.

```
fib(A,B,C)← C≤2, fib(A,B,C)← C≤1, fib(A,B,C)← C≤0, fib(A,B,C)← C≥0,
false(A)← A≤2, false(A)← A≤1, false(A)← A≤0, false(A)← A≥0,
```

Figure 11 shows the predicate dependency graphs before and after polyvariant specialisation using initial call $\texttt{false(A)} \leftarrow \texttt{A} \leq 2$, Ψ *shown above and an unfolding rule that just unfolds one step. Here,* `fib_1`, `fib_2` *and* `fib_3` *yield proof trees of dimension* ≤ 0, ≤ 1 *and* ≤ 2 *respectively.*

7 Discussion and related work

The control of partial evaluation, and more broadly of program specialisation, has been much studied [17]. The problem arises due to the two termination problems in specialisation: local termination, ensuring that loops in the program are not unfolded indefinitely, and global termination, which can be seen as the problem of generating only a finite number of versions of each program point. These two problems are mutually dependent: the more conservative the local unfolding strategy is (in order to ensure local

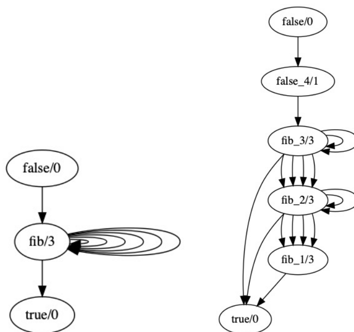

Figure 11: The predicate dependency graph for Example 5 producing clauses yielding proof trees of dimension at most 2, before and after polyvariant specialisation

termination), the more important it is to allow multiple versions of program points in order to preserve information, thus increasing the risk of global non-termination. In some approaches, the two problems are merged in order to handle these interactions [27, 25, 23, 20]. However, from the point of view of conceptual clarity and implementability, we argue that it is desirable to separate the two problems. Thus it becomes important to have a flexible way of controlling polyvariance and global termination.

The problem of polyvariant specialisation also occurs in offline specialisation; the unfolding of a program point marked as static by a binding time analysis must be accompanied by an assurance that only a finite number of static instances will arise during specialisation, and thus a finite number of versions of the program point will be generated. This is known as bounded static variation [17].

The algorithm presented here is based on a framework for partial evaluation of logic programs originally formulated in [2] and refined in [12], based on the framework presented by Lloyd and Shepherdson [22]. Some of the definitions in Section 2 are from these works, adapted for constrained Horn clauses.

The concept of property-based abstraction has been widely used in software model checking and was first introduced by Ball *et al.* [1] where it is called the Cartesian abstraction. It is used in a form similar to that shown in this paper in the HSF tool [15], though that work does not use the negations of the properties as we do (following [1]). Although we do not present an abstract interpretation [4], the domain of properties based on a finite set of constraints forms a lattice and the process of abstracting a concrete property is an example of a Galois connection. Being a finite domain, it has both advantages and disadvantages compared to other domains that could be used to control polyvariance. Property-based abstraction places a bound on the number of realisable versions, whereas with an infinite-height abstract domain, with global termination ensured by widening, the dependence of global termination on local unfolding would be loosened and an unbounded number of versions could be produced. A general presentation of the relation between fixed height versus infinite height domains can be found in [5], while an attempt to implement global control using an infinite height lattice is shown in [14].

Further research is needed on the automatic generation of properties. In the applications discussed in Section 6, properties were generated automatically using various heuristics.

Property-based abstractions are indirectly related to trace-based abstractions, which have been used in partial evaluation and supercompilation to control polyvariance, e.g. [27, 20, 13, 6]. Properties determine traces and vice versa; a property constrains the feasible program traces; whereas a trace implicitly defines properties which permit the trace (a principle explicitly used by de Angelis *et al.* [6]). We consider property-based abstractions to have some practical and conceptual advantages, opening up the use

of satisfiability solvers to compute the abstraction. Further evaluation and investigation on the choice of properties are needed.

The usefulness of specialisation as a component in program verification tools has been established in many works, including [28, 21, 7, 18] to name only a few. Fioravanti *et al.* investigated the trade-offs of polyvariance with efficiency and precision when using specialisation as a verification tool [11]. The use of constrained Horn clauses as a semantic representation formalism for verification of a wide range of languages and systems is now well established [15, 3]. Here, we emphasise the role of polyvariant specialisation in generating multiple versions of program points in a controlled way, when such versions lead to different control flow. This in turn implicitly allows disjunctive properties to be handled.

Acknowledgements. The author acknowledges very useful discussions with Bishoksan Kafle, Pierre Ganty, Samir Genaim and Jesús Doménech, contributing to the ideas presented here, as well as improvements suggested by the reviewers.

References

[1] T. Ball, A. Podelski & S. K. Rajamani (2001): *Boolean and Cartesian Abstraction for Model Checking C Programs*. In T. Margaria & W. Yi, editors: *Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001, Proceedings*, *Lecture Notes in Computer Science* 2031, Springer, pp. 268–283, doi:10.1007/3-540-45319-9_19.

[2] K. Benkerimi & J. W. Lloyd (1990): *A Partial Evaluation Procedure for Logic Programs*. In S. K. Debray & M. V. Hermenegildo, editors: *Logic Programming, Proceedings of the 1990 North American Conference*, MIT Press, pp. 343–358.

[3] N. Bjørner, A. Gurfinkel, K. L. McMillan & A. Rybalchenko (2015): *Horn Clause Solvers for Program Verification*. In L. D. Beklemishev, A. Blass, N. Dershowitz, B. Finkbeiner & W. Schulte, editors: *Fields of Logic and Computation II*, *LNCS* 9300, Springer, pp. 24–51, doi:10.1007/978-3-319-23534-9_2.

[4] P. Cousot & R. Cousot (1977): *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*. In: *Proceedings of the 4th ACM Symposium on Principles of Programming Languages, Los Angeles*, pp. 238–252, doi:10.1145/512950.512973.

[5] P. Cousot & R. Cousot (1992): *Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation*. In: *4th International Symposium on Programming Language Implementation and Logic Programming*, *Springer-Verlag Lecture Notes in Computer Science* 631, pp. 269–295, doi:10.1007/3-540-55844-6_142.

[6] E. De Angelis, F. Fioravanti, A. Pettorossi & M. Proietti (2012): *Specialization with Constrained Generalization for Software Model Checking*. In E. Albert, editor: *LOPSTR 2012*, *Lecture Notes in Computer Science* 7844, Springer, pp. 51–70, doi:10.1007/978-3-642-38197-3_5.

[7] E. De Angelis, F. Fioravanti, A. Pettorossi & M. Proietti (2014): *Program verification via iterated specialization*. *Sci. Comput. Program.* 95, pp. 149–175, doi:10.1016/j.scico.2014.05.017.

[8] E. De Angelis, F. Fioravanti, A. Pettorossi & M. Proietti (2018): *Predicate Pairing for program verification*. *TPLP* 18(2), pp. 126–166, doi:10.1017/S1471068417000497.

[9] J. Doménech, S. Genaim & J. P. Gallagher (2018): *Control-Flow Refinement via Partial Evaluation*. In S. Lucas, editor: *16th International Workshop on Termination (WST 2018)*, pp. 55–59. Available at `http://wst2018.webs.upv.es/wst2018proceedings.pdf`.

[10] S. Etalle & M. Gabbrielli (1996): *Transformations of CLP Modules*. *Theoretical Computer Science* 166, pp. 101–146, doi:10.1016/0304-3975(95)00148-4.

[11] F. Fioravanti, A. Pettorossi, M. Proietti & V. Senni (2013): *Controlling Polyvariance for Specialization-based Verification*. Fundam. Inform. 124(4), pp. 483–502, doi:10.3233/FI-2013-845.

[12] J. P. Gallagher (1993): *Specialisation of Logic Programs: A Tutorial*. In: Proceedings PEPM'93, ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, ACM Press, Copenhagen, pp. 88–98, doi:10.1145/154630.154640.

[13] J. P. Gallagher & L. Lafave (1996): *Regular Approximation of Computation Paths in Logic and Functional Languages*. In O. Danvy, R. Glück & P. Thiemann, editors: Partial Evaluation, Springer-Verlag Lecture Notes in Computer Science 1110, pp. 115–136, doi:10.1007/3-540-61580-6_7.

[14] J. P. Gallagher & J. C. Peralta (2001): *Regular Tree Languages as an Abstract Domain in Program Specialisation*. Higher-Order and Symbolic Computation 14(2-3), pp. 143–172, doi:10.1023/A:1012936614361.

[15] S. Grebenshchikov, N. P. Lopes, C. Popeea & A. Rybalchenko (2012): *Synthesizing software verifiers from proof rules*. In J. Vitek, H. Lin & F. Tip, editors: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, ACM, pp. 405–416, doi:10.1145/2254064.2254112.

[16] S. Gulwani, S. Jain & E. Koskinen (2009): *Control-flow refinement and progress invariants for bound analysis*. In M. Hind & A. Diwan, editors: PLDI 2009, ACM, pp. 375–385, doi:10.1145/1542476.1542518.

[17] N. D. Jones, C. Gomard & P. Sestoft (1993): *Partial Evaluation and Automatic Software Generation*. Prentice Hall, doi:10.1016/j.scico.2004.03.010.

[18] B. Kafle, J. P. Gallagher, G. Gange, P. Schachte, H. Søndergaard & P. J. Stuckey (2018): *An iterative approach to precondition inference using constrained Horn clauses*. TPLP 18(3-4), pp. 553–570, doi:10.1017/S1471068418000091.

[19] B. Kafle, J. P. Gallagher & P. Ganty (2018): *Tree dimension in verification of constrained Horn clauses*. TPLP 18(2), pp. 224–251, doi:10.1017/S1471068418000030.

[20] M. Leuschel & B. Martens (1996): *Global Control for Partial Deduction through Characteristic Atoms and Global Trees*. In O. Danvy, R. Glück & P. Thiemann, editors: Partial Evaluation, Springer-Verlag Lecture Notes in Computer Science 1110, pp. 263–283, doi:10.1007/3-540-61580-6_13.

[21] M. Leuschel & T. Massart (2000): *Infinite State Model Checking by Abstract Interpretation and Program Specialisation*. In A. Bossi, editor: Logic-Based Program Synthesis and Transformation (LOPSTR'99), Springer-Verlag Lecture Notes in Computer Science 1817, pp. 63–82, doi:10.1007/10720327_5.

[22] J. Lloyd & J. Shepherdson (1991): *Partial Evaluation in Logic Programming*. Journal of Logic Programming 11(3 & 4), pp. 217–242, doi:10.1016/0743-1066(91)90027-M.

[23] B. Martens & J. P. Gallagher (1995): *Ensuring Global Termination of Partial Deduction While Allowing Flexible Polyvariance*. In L. Sterling, editor: Proc. International Conference on Logic Programming, (ICLP'95), Tokyo, MIT Press.

[24] A. Pettorossi & M. Proietti (1999): *Synthesis and Transformation of Logic Programs Using Unfold/Fold Proofs*. J. Log. Program. 41(2-3), pp. 197–230, doi:10.1016/S0743-1066(99)00029-1.

[25] D. Sahlin (1993): *Mixtus: An Automatic Partial Avaluator for Full Prolog*. New Generation Comput. 12(1), pp. 7–51, doi:10.1007/BF03038271.

[26] R. Sharma, I. Dillig, T. Dillig & A. Aiken (2011): *Simplifying Loop Invariant Generation Using Splitter Predicates*. In G. Gopalakrishnan & S. Qadeer, editors: Computer Aided Verification, CAV 2011, Lecture Notes in Computer Science 6806, Springer, pp. 703–719, doi:10.1007/978-3-642-22110-1_57.

[27] V. Turchin (1988): *The Algorithm of generalization in the supercompiler*. In D. Bjørner, A. Ershov & N. Jones, editors: Proc. of the IFIP TC2 Workshop on Partial Evaluation and Mixed Computation, North-Holland, pp. 531–549.

[28] D. de Waal & J. P. Gallagher (1994): *The Applicability of Logic Program Analysis and Transformation to Theorem Proving*. In: Proceedings of the 12th International Conference on Automated Deduction (CADE-12), Nancy, doi:10.1007/3-540-58156-1_15.