Polyvariant program specialisation and its application in program analysis and verification

John P. Gallagher jpg@ruc.dk

Roskilde University, Denmark and IMDEA Software Institute, Madrid, Spain

Abstract. Specialisation is a program transformation that transforms a program with respect to given constraints that restrict its behaviour. Typically, the goal is to optimise a program, but specialisation can also transform programs to make them more amenable to analysis and verification. Here, we outline an algorithm for polyvariant specialisation of constrained Horn clauses (CHCs). Using CHCs as a representation language for imperative code, this has led to recent applications of the algorithm in termination and cost analysis, program verification and derivation of sufficient preconditions for program safety.

1 Polyvariant program specialisation

To specialise a program is to transform it by exploiting constraints that restrict its behaviour. The purpose of the work presented here is to specialise programs based on constraints *within* the program code, rather than constraints on the input, as is more typical. For example, when specialising a statement $\texttt{if}(e)\{s_1\}\{s_2\}$, assuming that 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 $\neg e$.

An essential feature of specialisation algorithms that can exploit internal constraints is *polyvariance*. This means that the transformation can produce two or more specialised *versions* of the same program code. For example suppose that the statement $\texttt{if}(e)\{s_1\}\{s_2\}$ is reached twice during a computation, once with the constraint c_1 such that c_1 implies e, and the other with the constraint c_2 which implies $\neg e$. Polyvariant specialisation allows two instances of the `if` statement to be generated in the specialised code, namely s_1 (specialised w.r.t. c_1) and s_2 (specialised w.r.t. c_2). 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.

Consider the fragment of imperative code shown in Figure 1, along with its representation as a set of constrained Horn clauses (CHCs). Due to interdependencies within the code, the **while** loop executes two distinct phases, the first executing only the "then" branch of the **if** statement and the second executing only the "else" branch. This leads to the polyvariant specialisation shown in Figure 2, again showing the imperative code, the corresponding CHCs and

the control-flow graph. In order to realise the two implicit loops, both internal specialisation and polyvariance are needed.

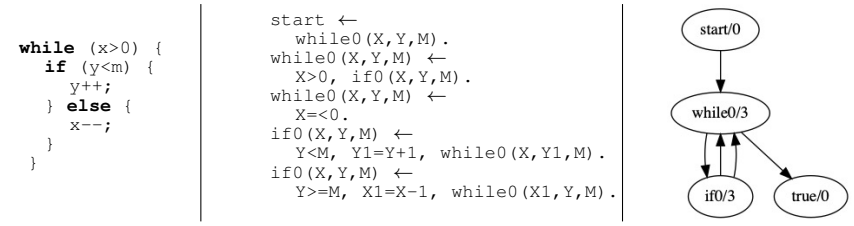

Fig. 1. An imperative code example, its CHC representation and its control-flow graph

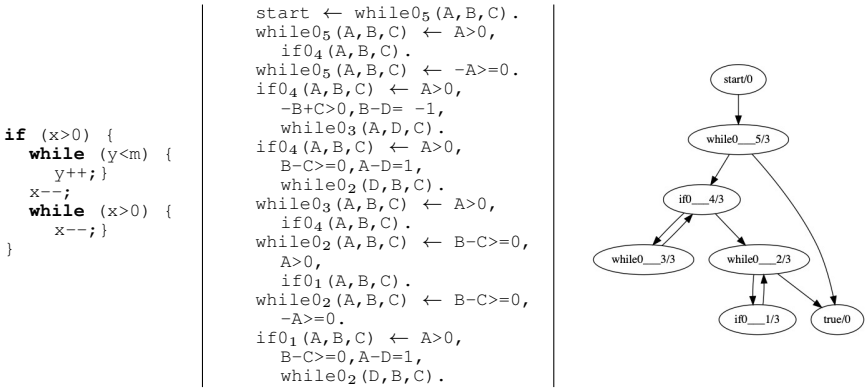

Fig. 2. Polyvariant specialisation of Figure 1

The advantage of the code in Figure 2 is that the loops are simpler. This means, for example, that simple linear ranking functions can be used to prove termination of the loops, whereas a more complex lexicographical ranking function is needed for the original loop in Figure 1. In general, experiments show that polyvariant specialisation can increase the ability of program analysis and verification tools to discover more precise code invariants.

2 Specialisation using property-based abstraction

In [9], an algorithm for polyvariant specialisation of CHCs is presented in detail. It is based on the "basic algorithm" in [7], which is parameterised by an unfolding operator and an abstraction operator. The abstraction is a *property-based*

```
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)
```

Fig. 3. Example from [13]: (left) original program, (right) translation to CHCs using backwards flow.

abstraction, drawing on an abstraction technique first introduced by Ball *et al.* [1] for software model checking, where it is called the Cartesian abstraction. The intuition behind the abstraction is that a finite set of properties of program variables is selected; the predicate calls that arise during specialisation are then abstracted according to which of the given properties (and their negations) are entailed by the call constraints.

For example, in Figure 2, polyvariant specialisation produces three versions of `while0`; the calls to $\texttt{while0}_2(\texttt{A},\texttt{B},\texttt{C})$ are those that entail $\texttt{B} \geq \texttt{C}$, the calls to $\texttt{while0}_3(\texttt{A},\texttt{B},\texttt{C})$ entail $\texttt{A} > \texttt{0}$ and the calls to $\texttt{while0}_5(\texttt{A},\texttt{B},\texttt{C})$ entail no properties. The set of chosen properties is critical to the quality of the specialisation. Essentially, the properties are derived from the tests in the program text.

3 Polyvariant specialisation in program analysis and verification

Pre-condition inference. In [13], specialisation 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.

Consider the example in Figure 3 taken from [13]. 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.

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 $\texttt{init}(\texttt{A},\texttt{B})$, namely $\texttt{B} \neq |\texttt{2A} - \texttt{100}|$ 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.

Termination analysis. In [5], 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

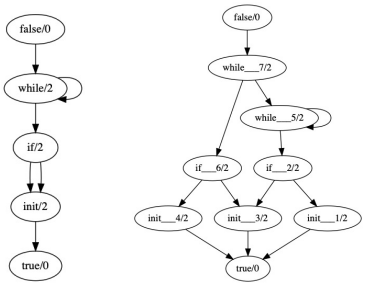

Fig. 4. The dependency graph for clauses in Figure 3, before and after polyvariant specialisation

of termination. The example from Figures 1 and 2 provides a typical example. 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 [5], showing that polyvariant specialisation of the control-flow graph very often improves the effectiveness of state-of-the-art tools for both automatic termination analysis and complexity bound analysis.

Dimension-based decomposition. The concept of *tree dimension* has been applied in verification to decompose a proof [14]. The dimension of a set of CHCs is a measure of their non-linearity. Given a set of CHCs for which some property is to be verified, we may decompose the problem by dimensions $0, 1, 2, \ldots$. Dimension-bounded sets of clauses can be generated using polyvariant specialisation. This enabled a variety of strategies for proof decomposition, described in [14].

Figure 5 shows the predicate dependency graphs for clauses defining the Fibonacci function before and after polyvariant specialisation. In the right hand figure, `fib_1`, `fib_2` and `fib_3` yield proof trees of dimension ≤ 0, ≤ 1 and ≤ 2 respectively.

4 Discussion and related work

The control of partial evaluation [12], and more broadly of program specialisation, has been much studied. A recurring problem is to allow sufficient polyvariance without endangering termination of the specialisation algorithm. The contribution of the present work is to use property-based abstraction to control polyvariance. 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. Further research is needed on the automatic generation of suitable sets of properties. In the applications discussed in Section 3, properties were generated automatically using various heuristics. We observed

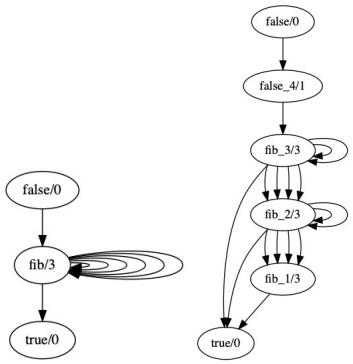

Fig. 5. The predicate dependency graph for a Fibonacci program producing clauses yielding proof trees of dimension at most 2, before and after polyvariant specialisation

a close relation between polyvariant specialisation using property-based abstract and *control-flow refinement* [11, 17].

Property-based abstractions are indirectly related to trace-based abstractions, which have been used in partial evaluation and supercompilation to control polyvariance, e.g. [18, 15, 8, 3]. 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.* [3]). 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 [19, 16, 4, 13] 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 [6]. 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 [10, 2]. 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 invariants to be derived.

Acknowledgements. The author acknowledges fruitful collaboration on applications of polyvariant specialisation with the co-authors of papers [14], [13] and [5]: (in alphabetical order) Jesús Doménech, Graeme Gange, Pierre Ganty, Samir Genaim, Bishoksan Kafle, Peter Schachte, Peter J. Stuckey and Harald Søndergaard.

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. 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.
3. 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.
4. 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. Available at https://doi.org/10.1016/j.scico.2014.05.017.
5. J. J. Doménech, J. P. Gallagher & S. Genaim (2019): *Control-Flow Refinement by Partial Evaluation, and its Application to Termination and Cost Analysis*. *TPLP* 19(5-6), pp. 990–1005, doi:10.1017/S1471068419000310. Available at https://doi.org/10.1017/S1471068419000310.
6. 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.
7. 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.
8. 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.
9. J. P. Gallagher (2019): *Polyvariant program specialisation with property-based abstraction*. In A. Lisitsa & A. P. Nemytykh, editors: *Proceedings Seventh International Workshop on Verification and Program Transformation (VPT-19)*, *EPTCS* 299, doi:10.4204/EPTCS.299.6.
10. 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.
11. 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.
12. N. D. Jones, C. Gomard & P. Sestoft (1993): *Partial Evaluation and Automatic Software Generation*. Prentice Hall, doi:10.1016/j.scico.2004.03.010.
13. 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.

14. 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.
15. 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.
16. 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.
17. 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.
18. 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.
19. 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.