An Experiment Combining Specialization with Abstract Interpretation

Gallagher, John Patrick; Glück, Robert

# An Experiment Combining Specialization
# with Abstract Interpretation

John P. Gallagher

Roskilde University, Denmark    IMDEA Software Institute, Spain

`jpg@ruc.dk`

Robert Glück

Copenhagen University, Denmark

`glueck@acm.org`

**Introduction and motivation.**    It was previously shown that control-flow refinement can be achieved by a program specializer incorporating property-based abstraction, as described in [6] and applied in [2] to improve termination and complexity analysis tools. We now show that this purpose-built specializer can be reconstructed in a more modular way, and that the previous results can be achieved using an off-the-shelf partial evaluation tool, applied to an abstract interpreter. The key feature of the abstract interpreter is the abstract domain, which is the product of the property-based abstract domain with the concrete domain. This language-independent framework provides a practical approach to implementing a variety of powerful specializers, and contributes to a stream of research on using interpreters and specialization to achieve program transformations.

**Abstract interpreters.**    Let $L$ be a programming language. We consider a program $p \in L$ to be a partial function of one argument (possibly an $n$-tuple), denoted $[\![p]\!]$, and assume that both argument and result are elements of a set $S$. An *interpreter* for $L$ is a program $I$ such that for all $p \in L$, $v \in S$, $[\![I]\!](p\ v) = [\![p]\!]\ v$ (or both are undefined). An *abstract interpreter* computes a *safe approximation* of an interpreter $I$. For the present discussion, we say that an abstract interpreter $A$ takes a program $p \in L$ with a *set* $\phi$ of input values, and computes a set of values as output. $A$ is a safe approximation of $I$ if for all $p \in L$ and $\phi \in \wp(S)$, $\{[\![I]\!](p\ v) \mid v \in \phi\} \subseteq [\![A]\!](p\ \phi)$. In other words, $A$ over-approximates the set of results that $I$ computes on elements of $\phi$.

In practice, abstract interpreters represent elements of $\wp(S)$ by descriptions in some *abstract domain* $D$. If $D$ is finite, an abstract interpreter using $D$ is a total function $[\![A]\!] : (L \times D) \to D$, that is, $[\![A]\!](p\ \phi)$ terminates for all $p \in L$ and $\phi \in D$. Although abstract interpreters are typically designed to terminate, domains that are infinite and abstract interpretations that do not guarantee termination are also useful; we can still gain interesting information from running them. In what follows, we define a *mixed* abstract interpreter that combines the concrete domain $\wp(S)$ with a finite domain $D$. In the theory of abstract interpretation [1], the Cartesian product $\wp(S) \times D$ is an abstract domain. The abstract interpreter $A$ used in our experiment has type $(L \times ((\wp(S) \times D))) \to (\wp(S) \times D)$.

**Structure of an interpreter.**    Let us assume that an (abstract) interpreter operates as a transition system, though this is not essential. A state consists of a point in the program being interpreted together with the values of variables in the domain of interpretation at that point. For the standard interpreter, let $q$ and $q'$ be program points, and $v, v' \in S$ be the respective values of the variables at those points. A transition is $\langle q, v \rangle \xrightarrow{\delta(v,q)=v'} \langle q', v' \rangle$, where $\delta$ is a function relating $v$ and $v'$ at the point $q$. A transition in an abstract interpreter over domain $D$ uses a mapping $\delta_D(\phi, q) = \phi'$ where $\phi, \phi' \in D$ ($\delta_D$ is sometimes called the abstract "transfer function").

Transitions in an abstract interpreter over the product domain $\wp(S) \times D$ have two components corresponding to $\wp(S)$ and $D$ respectively: they have the form $\langle q, \{v\}, \phi \rangle \xrightarrow{\delta(v,q)=v'; \delta_D(\phi,q)=\phi} \langle q', \{v'\}, \phi' \rangle$,

where $\delta$ and $\delta_D$ are the transfer functions for $\wp(S)$ and $D$ respectively. Assuming that $\delta$ is the standard interpreter transfer function, this both computes the standard result as well as an abstract result in $D$. We will see that we can exploit the separate components during specialization. We assume that the initial call to the interpreter contains a singleton set $\{v\} \in \wp(S)$, and thus only singleton concrete states are reachable.

**Specialization.** A *specializer* for $L$ is a program $S$ that transforms a program $p \in L$ with respect to partially specified input. We assume that $p$'s argument is a pair $(v_1 \ v_2)$ and that $S$ is provided with $v_1$. The result is a program $p' \in L$, i.e. $[\![S]\!](p \ v_1) = p'$ which satisfies the property $[\![p']\!] v_2 = [\![p]\!](v_1 \ v_2)$. Specialization of an interpreter with respect to a program in $L$ is known as the *first Futamura projection* [4, 11]. We have $[\![S]\!](I \ p) = I_p$, where according to the properties of interpreters and specializers, $[\![I_p]\!]v = [\![I]\!](p \ v) = [\![p]\!]v$. The program $I_p$ can be seen as the compilation or transformation of $p$ into the language of $I$. Values encountered during specialization are *static* or *dynamic*, in the terminology of partial evaluation. Functions with static arguments can be evaluated during specialization, while functions with dynamic arguments are not, and are retained in the specialised program. A *binding-time analysis* [11] can determine which parts of the program to be specialised are guaranteed to be static.

**Abstract interpreter specialization.** Consider the specialization of an abstract interpreter with domain $\wp(S) \times D$. In a state $\langle q, \{v\}, \phi \rangle$, we can determine that the program point $q$ and the abstract state $\phi$ are static, while $v$ is dynamic. This is because the initial abstract state is static (even if it is the "top" element of $D$) and in a transition from $\langle q, \{v\}, \phi \rangle$, where $q$ and $\phi$ are static, and $v$ dynamic, $\delta_D(\phi, q) = \phi'$ can be evaluated while $\delta(v, q) = v'$ cannot; thus the computation $\delta(v, q) = v'$ is retained in the residual specialised interpreter. Thus in the next state $\langle q', \{v'\}, \phi' \rangle$, $q'$ and $\phi'$ are static, while $v'$ is dynamic. Furthermore, if $D$ is finite, then the static values have *bounded static variation*, which means that only a finite number of different values of the static arguments arise during specialization. This leads to a so-called *polyvariant* specialization. A transition of the form $\langle q, \{v\}, \phi \rangle \xrightarrow{\delta(v,q)=v'; \delta_D(\phi,q)=\phi'} \langle q', \{v'\}, \phi' \rangle$ is specialised into a finite number of transitions of the form $\langle q_\phi, v \rangle \xrightarrow{\delta(v,q)=v'} \langle q'_{\phi'}, v' \rangle$, for each pair $(q, \phi)$ encountered during specialization.

**Control-flow refinement by mixed interpreter specialization.** A abstract interpreter for constrained Horn clauses was written as a Prolog program[1]. The abstract domain is a product domain as described above, where $D$ is a set $\wp(\Psi)$ where $\Psi$ is finite set of properties. The main interpreter predicate is `solve(Q,A,Phi,Psi,Prog)`, representing a state of the interpreter with a call to predicate `Q` with concrete arguments `A`, abstract state (the set of properties `Phi` from `Psi` that `A` entails), `Psi` and `Prog`, the last two being the set of all properties and the set of Horn clauses respectively. A transition of the interpreter evaluates two calls `delta` and `delta_D`, corresponding to $\delta$ and $\delta_D$ above. $\delta$ evaluates the constraints of the clauses, and the $\delta_D$ computes the properties for the body calls in the clause. When run normally, the interpreter mirrors the standard semantics, but in addition carries around the set of properties that hold. Consider the following example clauses considered in [6].

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

The interpreter applied to these clauses can run a goal of the form `solve(while0(5,3,10),....)` and terminate.

The offline partial evaluator LOGEN [14] was used to partially evaluate the interpreter with respect to a set of clauses and a fixed finite set of properties. To use LOGEN, each call in the interpreter is annotated as *unfold* or *memo*, and each argument of memoed calls is annotated as *static*, *dynamic* or *nonvar* (meaning that everything below the top level of the term is dynamic). In the interpreter sketched

---

[1]Available at https://github.com/jpgallagher/absint4pe

above, the calls to `solve` and `delta` are memoed, while all other calls, including `delta_D`, are unfolded. The arguments `Q`, `Phi`, `Psi` and `Prog` are *static*, while `A` is *nonvar*. The specialised program thus consists solely of specialised clauses for `solve` and the concrete constraints linking one concrete state with the next.

**Example result.** The result of specialization of the clauses above, using the same set of properties in the `Psi` argument as were used in [6] is as follows.

```
solve__2(A,B,C) :- A>0,
   solve__3(A,B,C).                solve__4(A,B,C) :- A>0, solve__3(A,B,C).
solve__2(A,B,C) :- A=<0.          solve__5(A,B,C) :- A>0,B>=C,
solve__3(A,B,C) :-B<C,A>0,           solve__6(A-1,B,C).
   solve__4(A,B+1,C).             solve__5(A,B,C) :- A=<0.
solve__3(A,B,C) :- B>=C,          solve__6(A,B,C) :- B>=C,D=A-1,
   solve__5(A-1,B,C).                solve__5(D,B,C).
```

This result is identical, apart from predicate names, to the result obtained in [6]. Polyvariance is exemplified by `solve_2`, `solve_4` and `solve_5`, these being three versions of calls to `solve` when interpreting a call to `while0` in the input clauses, corresponding to different values for the static arguments (the properties that hold) arising during partial evaluation. Similarly, `solve_3` and `solve_6` are versions of the interpretation of `if0`.

The implementation of `delta_D` reused code from the specializer described in [6], but the abstract interpreter has a simpler structure than the specializer used in that work. This is due to the fact that the operations handling unfolding, memoing, generalization and polyvariance are handled by LOGEN and do not need to be included in the interpreter. Furthermore, it would be simple to replace the code for `delta_D` with an implementation of an abstract transfer function for some other domain.

**Related and future work.** The transformation of programs by specialising interpreters goes back to the Futamura projections [4]. The projections can be exploited by inserting more sophisticated interpreters between a program and the specializer (e.g.[5, 16, 9, 12]). The power of the overall program transformation has been improved by combining specialization with abstraction [10, 15, 13, 3]. The main contrast to previous work on combining specialization with abstract interpretation is that we choose not to integrate abstract interpretation in the specializer, but into the interpreter. Thus a simple partial evaluator (in our case LOGEN can achieve the same results as the more elaborate specializers incorporating abstract interpretation. We argue that the approach of combining the interpretive approach with an abstract interpreter has practical advantages such as modularity and ease of implementation. The same transformation power of a sophisticated specializer can be achieved by interpreter specialization provided the underlying specializer is Jones-optimal and performs static expression reduction [8]. Often it is easier to modify an interpreter than the specialization tool. Also, it only requires to reason about the correctness of the interpreter provided the underlying specializer is correct.

The approach presented here needs further research. For instance, interpreters may be parameterised by abstract interpretation domains; the 'binding-time improvement' of the interpreter is thereby done only once. The approach is not limited to offline specialization; other specialization tools such as online specializers and supercompilers may be used. Clearly, the interpretive approach lends itself to generate specializers by the specializer projections [7]. These will be challenges for further investigations.

# References

[1] P. Cousot & R. Cousot (1977): *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints.* In: *POPL*, pp. 238–252, doi:`10.1145/512950.512973`.

[2] 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`.

[3] 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`.

[4] Y. Futamura (1971): *Partial Evaluation of Computation Process - An Approach to a Compiler-Compiler*. *Systems, Computers, Controls* 2(5), pp. 45–50.

[5] J. P. Gallagher (1986): *Transforming Logic Programs by Specialising Interpreters*. In: *Proceedings of the 7th European Conference on Artificial Intelligence (ECAI-86), Brighton*, pp. 109–122.

[6] J. P. Gallagher (2019): *Polyvariant program specialisation with property-based abstraction*. In A. Lisitsa & A. P. Nemytykh, editors: *VPT-19, EPTCS* 299, doi:`10.4204/EPTCS.299.6`.

[7] R. Glück (1994): *On the generation of specializers*. *Journal of Functional Programming* 4(4), pp. 499–514, doi:`10.1017/S0956796800001167`.

[8] R. Glück (2002): *Jones Optimality, Binding-Time Improvements, and the Strength of Program Specializers*. In: *Proc. Asian Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, ACM, pp. 9–19, doi:`10.1145/568173.568175`.

[9] R. Glück & J. Jørgensen (1994): *Generating transformers for deforestation and supercompilation*. In B. Le Charlier, editor: *Static Analysis. Proceedings*, LNCS 864, Springer-Verlag, pp. 432–448, doi:`10.1007/3-540-58485-4_57`.

[10] J. Hatcliff, M. Dwyer & S. Laubach (1998): *Staging static analyses using abstraction-based program specialization*. In C. Palamidessi et al., editors: *Principles of Declarative Programming*, LNCS 1490, Springer, pp. 134–151, doi:`10.1007/BFb0056612`.

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

[12] N. D. Jones (2004): *Transformation by interpreter specialization*. *SCP* 52(1-3), pp. 307–339, doi:`10.1016/j.scico.2004.03.010`.

[13] M. Leuschel (2004): *A framework for the integration of partial evaluation and abstract interpretation of logic programs*. *ACM TOPLAS* 26(3), pp. 413–463, doi:`10.1145/982158.982159`.

[14] M. Leuschel, D. Elphick, M. Varea, S. Craig & M. Fontaine (2006): *The Ecce and Logen partial evaluators and their web interfaces*. In J. Hatcliff & F. Tip, editors: *PEPM*, ACM, pp. 88–94, doi:`10.1145/1111542.1111557`.

[15] G. Puebla, M. Hermenegildo & J. P. Gallagher (1999): *An integration of partial evaluation in a generic abstract interpretation framework*. In O. Danvy, editor: *PEPM'99*, San Antonio, Texas, pp. 75–84.

[16] V. F. Turchin (1993): *Program transformation with metasystem transitions*. *Journal of Functional Programming* 3(3), pp. 283–313, doi:`10.1017/S0956796800000757`.