Abstract Partial Deduction using Regular Types and its Application to Model Checking

(Extended Abstract)

Michael Leuschel and Stefan Gruner
Declarative Systems and Software Engineering
Department of Electronics and Computer Science
University of Southampton
Southampton SO17 1BJ, UK

1 Introduction

Program specialisation, also called *partial evaluation* or *partial deduction*, is an automatic technique for program optimisation (see, e.g., [10, 16, 20]). The central idea is to specialise a given source program for a particular application domain. The main idea of *abstract interpretation* [4, 2, 14] is to statically analyse programs by interpreting them over some *abstract domain*. This is done in such a way as to ensure termination of the abstract interpretation and to ensure that the so derived results are a *safe approximation* of the programs' concrete runtime behaviour(s).

At first sight *abstract interpretation* and *program specialisation* might appear to be quite different techniques: abstract interpretation focusses on *correct and precise* analysis, while the main goal of program specialisation is to produce more *efficient residual code* (for a given task at hand). Nonetheless, it is often felt that there is a close relationship between abstract interpretation and program specialisation and, recently, there has been a lot of interest in the integration of these two techniques (see, e.g., [3, 22, 15, 31]).

[19, 21] presented a generic framework and algorithm for top-down program specialisation, which supersedes earlier top-down approaches in generality and power. [19, 21] established several *generic correctness results* and have cast several existing techniques in the framework. [19, 21] has illustrated on some examples how the additional power can be exploited in practice, for improved generalisation, unfolding and code-generation. Applications in verification and model checking were also hinted at.

In this paper we present a particular instantiation of the abstract partial deduction framework of [19, 21], by using an abstract domain based upon regular types. This instantiation partly resembles the technique of [13] (the 2 techniques were developed more or less independently). We describe our implementation, based upon the ECCE system [18]. We also present the very first experiments with this implementation and demonstrate its usefulness for infinite state model checking of process algebras. The final version of the paper will contain a full empirical study.

2 Preliminaries

In the remainder of this paper we suppose some familiarity with basic notions in logic programming [1, 27] as well as some acquaintance with partial deduction [20]. In this paper, we restrict ourselves to definite programs and goals (but possibly with declarative built-in's such as is, $call$, $functor$, arg, $\backslash{==}$).

We need the following definitions. An *expression* is either a term, an atom or a conjunction. We use $E_1 \preceq E_2$ to denote that the expression E_1 is an instance of the expression E_2. By $vars(E)$ we denote the set of variables appearing in an expression E. By *mgu* we denote a function which computes an idempotent and relevant[1] most general unifier θ of two expressions E_1 and E_2 (and returns *fail* if no such unifier exists) Also, by $msg(E_1, E_2)$ we denote a most specific generalisation [17] of E_1 and E_2.

Also, as common in partial deduction, the notion of SLD-trees is extended to allow *incomplete* SLD-trees which may contain leaves where no literal has been selected for further derivation. A *resultant* is a formula $H \leftarrow B$ where H and B are conjunctions of literals. The resultant of a SLD-derivation of $P \cup \{\leftarrow Q\}$ with c.a.s. θ leading to the *resolvent* $\leftarrow B$ is the formula $Q\theta \leftarrow B$. We also denote by $\mathcal{Q}$ the set of all conjunctions and by 2^S the power-set of some set S.

Abstract Partial Deduction In classical partial deduction a program is specialised for a set of ordinary atoms (or conjunctions in the case of conjunctive partial deduction). In abstract partial deduction a program is specialised for a set of abstract conjunctions, defined as follows:

Definition 2.1 An *abstract domain* $(\mathcal{AQ}, \gamma)$ is a couple consisting of a set $\mathcal{AQ}$ of so-called *abstract conjunctions* and a total *concretisation function* $\gamma : \mathcal{AQ} \mapsto 2^{\mathcal{Q}}$, providing the link between the abstract and the concrete domain, such that:

1. $Q \in \gamma(\mathbf{A}) \;\Rightarrow\; \forall\theta : Q\theta \in \gamma(\mathbf{A})$,
2. $\forall \mathbf{A} \in \mathcal{AQ} \quad \exists Q \in \mathcal{Q}$ such that $Q' \in \gamma(\mathbf{A}) \Rightarrow Q' \preceq Q$.

Property 1. expresses the requirement that the image of $\gamma(.)$ is *downwards closed*, which is vital in correctness proofs [21]. Property 2. expresses the fact that all conjunctions in $\gamma(\mathbf{A})$ have the same number of conjuncts and with the same predicates at the same position. This property is crucial to enable the construction of (correct) residual code. We will denote the fact that $\gamma(\mathbf{A_1}) \subseteq \gamma(\mathbf{A_2})$ by $\mathbf{A_1} \sqsubseteq \mathbf{A_2}$.

One particular abstract domain, which arises in the formalisation of (classical) partial deduction [28], is the $\mathcal{PD}$-domain where $\mathcal{AQ} = \mathcal{Q}$ (i.e. the abstract

[1] I.e., $\theta\theta = \theta$ and $vars(\theta) \subseteq vars(E_1) \cup vars(E_2)$.

conjunctions are the concrete ones) and $\gamma(Q) = \{Q' \mid Q' \preceq Q\}$ (i.e. an abstract conjunction denotes all its instances). For example, we can use the concrete conjunction $p(X) \wedge q(X)$ as an abstract conjunction in the $\mathcal{PD}$-domain with $p(a) \wedge q(a) \in \gamma(p(X) \wedge q(X))$ as well as $p(X) \wedge q(X) \in \gamma(p(X) \wedge q(X))$, but $p(a) \wedge q(b) \notin \gamma(p(X) \wedge q(X))$.

The main difference of abstract partial deduction [21] over abstract interpretation is that both the concrete and the abstract domain have to be analysed (the residual program has to be expressed in the concrete domain). Concretely, when specialising an abstract conjunction $\mathbf{A}$ we have to generate:

1. *concrete* resultants $H_i \leftarrow B_i$, which have to be totally correct for all possible calls in $\gamma(\mathbf{A})$,
2. for each resultant $H_i \leftarrow B_i$ an *abstract* conjunction $\mathbf{A_i}$ approximating all the possible resolvent goals which can occur after resolving an element of $\gamma(\mathbf{A})$ with $H_i \leftarrow B_i$.

Step 1. is called *abstract unfolding* and step 2. *abstract resolution*. The formal definitions can be found in [21]. An abstract partial deduction algorithm can now be obtained [21] by computing the least fixpoint of $U(S) = \omega(S \cup \{\mathit{ares}(\mathbf{A}, C) \mid \mathbf{A} \in S \wedge C \in \mathit{aunf}(\mathbf{A})\})$, where ω is a widening function ensuring termination. Thorugh this we obtain a set of abstract conjunctions from which we generate residual code using abstract unfolding combined with a renaming operation (which maps conjunctions back to atoms and thus general resultants back to Horn clauses).

3 Limitations of the $\mathcal{PD}$-domain for Verification

Let us first present a verification example where existing partial deduction techniques, i.e., those based on the $\mathcal{PD}$-domain, fail. This motivates the development of a more precise domain later in the paper.

Example 3.1 Take the following specification of an agent P expressed using the process algebra CSP [33] (where "$\rightarrow$" denotes the action prefix, and "$|||$" the interleaving operator): $P =_{Def} a \rightarrow (P|||P)$
The operational semantics of CSP tells us that the agent P can perform the action a (leading to the expression $P|||P$). This expression, in turn, can perform a leading to either $(P|||P)|||P$ or $P|||(P|||P)$, and so on. This CSP specification is thus infinite state, in the sense that an infinite number of different expressions can be reached. Infinite state descriptions can be very useful (e.g., to specify protocols with an arbitrary number of connections) but existing tools (such as FDR [33]) cannot cope with them and are thus, e.g., unable to establish that P cannot perform the action b.

In recent work it has been shown [26] that infinite state model checking can be performed by a combination of partial deduction of logic programs and abstract interpretation. In that approach, one would first encode P and the safety property that no b actions should ever be possible, as a logic program. This can for example be done as follows:

```
unsafe :- unsafe(agent(p)).
unsafe(State) :- trans(b,State,_).
unsafe(State) :- trans(Action,State,NewState),unsafe(NewState).
trans(A,prefix(A,X),X).
trans(A,par(X,Y),par(X1,Y)) :- trans(A,X,X1).
trans(A,par(X,Y),par(X,Y1)) :- trans(A,Y,Y1).
trans(A,agent(X),X1) :- agent(X,XDef),trans(A,XDef,X1).
agent(p,  prefix(a, par(agent(p),agent(p)) )).
```

To prove our safety property, [26] then proposes to use partial deduction combined with abstract interpretation to attempt to transform the above program into the empty program. If this transformation is successful, the infinite state system is safe. This approach can successfully handle [24] various versions and extensions of Petri nets. However, the state expressions of process algebras are more involved and existing partial deduction techniques will not be able to deduce that `unsafe` must fail. Let us examine why this is the case. First, as explained above, the state P leads to the state $P|||P$, and hence the call

`unsafe(agent(p))` will lead to `unsafe(par(agent(p),agent(p)))`.

At that point most partial deduction systems will stop the unfolding[2] and perform generalisation.[3] Now, in the $\mathcal{PD}$-domain the most precise generalisation possible is the *msg* of the two calls which in this case is `unsafe(X)`. In other words, we lose all the information on the system and we cannot prove the safety property. Indeed, the unconstrained call `unsafe(X)` can of course also represent `unsafe(prefix(b,Z))` which *can* perform the action b. Hence, it is impossible to transform `unsafe(X)` into *fail*.

4 The RUL Domain

To solve the above problem (and many others as well), we propose to move to a more precise abstract domain in which we can implement a more refined generalisation operation. More concretely, we propose to use an abstract domain using regular types encoded as regular unary logic programs [34, 12], for which efficient algorithms exist and which is nonetheless a sufficiently expressive domain. We believe that this domain will also yield improved precision in more "classical" applications of partial deduction. We will use the following (adapted) definition:

Definition 4.1 A *canonical regular unary clause* is a clause of the form:

$$t_0(f(X_1,\ldots,X_n)) \leftarrow t_1(X_1) \wedge \ldots t_n(X_n)$$

where $n \geq 0$ and $X_1,\ldots,X_n$ are distinct variables. A *canonical regular unary logic (RUL) program* is a finite set of regular unary clauses, in which no two different clause heads have a common instance, together with the single fact $any(X) \leftarrow$. Given a (possibly non-ground) conjunction T, we write $R \models \forall(T)$ iff $R \cup \{\leftarrow T\}$ has an SLD-refutation with the empty computed answer.

[2] E.g., ECCE will detect that `agent(p)` $\trianglelefteq$ `par(agent(p),agent(p))`.

[3] A similar situation will arise even if we delay generalisation.

Example 4.2 For example, given the following RUL-program R, we have $R \models \forall(t1([a]))$ and $R \models \forall(t1([X,Y]))$.

```
t1([]).                          any(X).
t1([H|T]) :- any(H),t1(T).
```

Given two RUL-programs R_1, R_2, there exist efficient procedures for checking inclusion ($R_1 \subseteq R_2$), computing the intersection ($R_1 \cap R_2$) and computing an upper bound ($R_1 \sqcup R_2$). Note, the definitions of [34, 12] do not allow the *any* type. However, this extension does not cause any practical problems (indeed, the implementation discussed in [34] allows the *any* type; so does the tool [11] based upon [12]).

Definition 4.3 We define the *RUL-domain* $(\mathcal{AQ}, \gamma)$ to consist of abstract conjunctions of the form $\langle Q, T, R\rangle \in \mathcal{AQ}$ where Q, T are concrete conjunctions and R is a RUL program such that: $T = t_1(X_1) \wedge \ldots t_n(X_n)$, where $vars(Q) = \{X_1, \ldots, X_n\}$ and t_i are predicates defined in R. The concretisation function γ is defined as follows: $\gamma(\langle Q, T, R\rangle) = \{Q\theta \mid R \models \forall(T\theta)\}$.

The following is used to simplify type conjunctions and detect inconsistencies.

Definition 4.4 Simplification of a type conjunction T into T' within RUL program R, denoted by $T \rightsquigarrow_R T'$, is defined as follows. We build an SLD-tree τ for $R \cup \{\leftarrow T\}$ by repeatedly selecting the leftmost literal of the form $t_i(s_i)$, where $t_i \neq any$ and s_i is not a variable, until we only have calls with variables remaining. Due to the nature of R this SLD-tree will have a single branch with the empty computed answer. If this produces a finitely failed SLD-tree τ, we write $T \rightsquigarrow_R fail$, otherwise we write $T \rightsquigarrow_R T'$, where T' is the conjunction in the leaf of τ.

E.g., using R from Ex. 4.2, $t1([X, Y|T])) \rightsquigarrow_R any(X) \wedge any(Y) \wedge t1(T)$ and $t1(b) \rightsquigarrow_R fail$. Note that, by definition, $T \rightsquigarrow_R fail$ implies that $\forall\sigma.R \not\models \forall(T\sigma)$.

We can now define an abstract unfolding and an abstract resolution in the RUL-domain. First we compute the abstract unfolding of an abstract conjunction $\mathbf{A} = \langle Q, T, R\rangle$, denoted by $aunf(\mathbf{A})$, as follows:

1. do a classical unfolding of Q in P, producing an SLD-tree τ for $P \cup \{\leftarrow Q\}$ with resultants $\{Q\theta_1 \leftarrow B_1, \ldots, Q\theta_k \leftarrow B_k\}$.
2. For every resultant $Q\theta_i \leftarrow B_i$, compute $T\theta_i \rightsquigarrow_R T'$. If $T' = fail$ then we remove the resultant.

Thus, abstract unfolding is just like ordinary unfolding except that we prune all the branches which are incompatible with the regular type information. Take for example an abstract goal $\mathbf{A} = \langle p(X), t_1(X), R\rangle$ where R is taken from Ex. 4.2, and take the following program P:

```
p(a).
p([]).
p([H|T]) :- p(H),p(T).
```

Classical unfolding of $p(X)$ could give us (depending on the unfolding rule) the original program P back. As we have $t_1(a) \leadsto_R fail$ the first resultant will be pruned by the abstract unfolding of $\mathbf{A}$, which thus returns:

```
p([]).
p([H|T]) :- p(H),p(T).
```

To compute the abstract resolution of an abstract conjunction $\mathbf{A} = \langle Q, T, R \rangle$ and a particular resultant $R = Q\theta_i \leftarrow B_i$ in $aunf(\mathbf{A})$, denoted by $ares(\mathbf{A}, Q\theta_i \leftarrow B_i)$, we do the following:

3. Starting from T' obtained by $T\theta_i \leadsto_R T'$, we compute T'' as follows:
   - remove any $t(X)$ calls in T' for which $X \notin vars(B_j)$ (i.e., we do a projection).
   - As we may have multiple types for some variables (e.g., because of bindings such as $\theta_1 = \{X_1/X_2\}$), we compute the type intersection for these variables. Note that this step might modify the RUL-program R (new type predicates might have to be introduced), yielding a new RUL-program R' (if no changes are required, set $R' = R$).
   - Finally, we add $any(Y)$ to the disjunction for any fresh, existential variable Y in B_i for which no conjunct $t(Y)$ exists in T'.

   We return the abstract conjunction $\langle B_i, T'', R' \rangle$.

Let us return to our small example above, involving the abstract goal $\mathbf{A} = \langle p(X), t_1(X), R \rangle$, and let us compute $ares(\mathbf{A}, C)$, where $C = p([H|T]) \leftarrow p(H) \wedge p(T)$ is the second clause of the abstract unfolding. We have that $t_1([H|T]) \leadsto_R any(H) \wedge t_1(T)$ and thus $ares(\mathbf{A}, C) = \langle p(H) \wedge p(T), any(H) \wedge t_1(T), R \rangle$. The framework of [21] allows us to split this abstract conjunction into two abstract conjunctions $\mathbf{A_1} = \langle p(H), any(H), R \rangle$ and $\mathbf{A_2} = \langle p(T), t_1(T), R \rangle$, which can be specialised separately. We have that $\mathbf{A_2} \sqsubseteq \mathbf{A}$, and it does not have to be specialised, while $\mathbf{A_1} \not\sqsubseteq \mathbf{A}$ and we thus do have to specialise it. The final specialised program could thus look like (where renaming has mapped $\mathbf{A}$ to p_1 and $\mathbf{A}_2$ to p_2):

```
p_1([]).                          p_2(a).
p_1([H|T]) :- p_2(H),p_1(T).      p_2([]).
                                  p_2([H|T]) :- p_2(H),p_2(T).
```

Example 3.1 (continued) Using the RUL-domain, we can now solve the problem in Example 3.1. E.g., instead of producing the rough generalisation `unsafe(S)` we can produce $\mathbf{U} = \langle$`unsafe(S),t1(S),`$R\rangle$, where R is the following RUL-program:

```
t1(agent(X)) :- t2(X).            t2(p).
t1(par(X,Y)) :- t1(X),t1(Y).      any(X).
```

When unfolding the concrete call `unsafe(S)` of $\mathbf{U}$, we might (depending on the particular unfolding) get the following resultants R' and R'':

```
unsafe(S) :- trans(b,S,S2).
unsafe(S) :- trans(A,S,S2),unsafe(S2).
```

Let us examine the first resultant R'. For this resultant, we have $ares(\mathbf{U}, R') = \langle$`trans(b,S,S2),t1(S)`$\wedge$`any(S2)`$, R\rangle = \mathbf{T}$ (S simply gets the type from $\mathbf{U}$ and $S2$ is fresh and unconstrained). Let us examine this abstract conjunction $\mathbf{T}$ in more detail, and see whether our technique is powerful enough to infer that we cannot execute the action b. When unfolding the concrete part `trans(b,S,S2)` of $\mathbf{T}$ we might get the following (complete) list of resultants $\{R_1, R_2, R_3\}$:

```
trans(b,prefix(b,X),X).
trans(b,par(X,Y),par(X1,Y)) :- trans(b,X,X1).
trans(b,par(X,Y),par(X,Y1)) :- trans(b,Y,Y1).
```

Observe that the resultant R_1 will be pruned by the type information, because $t1(S)\{S/prefix(b,X)\} \rightsquigarrow_R fail$. Also, there is no resultant associated with clause 4 of `trans/3` because the call `trans(b,prefix(a,par(agent(p),agent(p))),S2)` fails finitely. Clauses R_2 and R_3 cannot be pruned by the type information, as $t1(S)\{S/par(X,Y)\} \rightsquigarrow_R t1(X) \wedge t1(Y)$.

For resultant R_2 we have $t1(par(X,Y)) \wedge any(par(X1,Y) \rightsquigarrow_R t1(X) \wedge t1(Y) \wedge any(X1) \wedge any(Y)$. Abstract resolution thus gives us $ares(\mathbf{T}, R_1) = \mathbf{T}' = \langle trans(b,X,X1), t1(X) \wedge any(X1), R\rangle$ for which $\mathbf{T}' \sqsubseteq \mathbf{T}$ (and $\mathbf{T} \sqsubseteq \mathbf{T}'$). A similar picture holds for resultant R_3. After renaming, mapping $\mathbf{T}$ to, e.g., $tb(A,S,S2)$, we thus obtain:

```
tb(b,par(X,Y),par(X1,Y)) :- tb(b,X,X1).
tb(b,par(X,Y),par(X,Y1)) :- tb(b,Y,Y1).
```

In the context of [26] we would run a bottom-up abstract interpretation (e.g., [29]) after the top-down specialisation to obtain: `tb(b,S,S2) :- fail.`

We have thus successfully established that the generalisation $\mathbf{U}$ cannot directly perform the action b. A similar (but somewhat more involved) picture will hold for the other resultants of $\mathbf{U}$ and, contrary to earlier techniques, we are able to infer that our system is safe using abstract partial deduction in the RUL-domain!

The next step is now to see whether the above approach can be fully automated. The difficult part will of course be handling the control, in particular the generalisation operation ($aunf(.)$ and $ares(.)$ have already been fully defined above). We describe our first attempts and our first implementation below.

5 Implementation in ECCE

The ECCE system [18] provides a highly parametric partial deduction procedure (cf. Appendix A) where one can dynamically change the parameters and even provide his own code. We have used this aspect of ECCE to encode an abstract conjunction $\mathbf{A} = \langle Q, T, R\rangle$ as a concrete conjunction $Q, rul\_constraint(T, R)$ and provided special support for the predicate $rul\_constraint/1$ using the toolkit [11, 12] by John Gallagher. For example, the *covered* check was re-implemented to take the type information into account. For this, goals are put into a normal form where all the synactic information is put in the regular types. E.g., $p(a)$

is normalised into $p(X), t_1(X)$ with RUL theory $t_1(a) \leftarrow$. This allows *covered* to be implemented using the inclusion test on regular types of [11].

For local control, we use determinate unfolding and the homeomorphic embedding $\trianglelefteq$ augmented with a satsifiabilty check on the type information (which is currently applied after every unfolding step). Otherwise, no special treatment for regular types have to be provided as they will never be unfolded by ECCE directly.

Concerning the global control, the simplest approach is to use a $\mathit{whistle}(L, \gamma)$ which blows whenever two calls have the same predicate and then use a generalisation operation $\mathit{generalise}(L, W, \gamma)$ which computes an upper bound $\sqcup$ and then shortens the obtained RUL theory using the algorithms of [12]. This implements a mono-variant specialisation technique which disregards the regular type information within the *whistle*. A more refined *whistle* can be obtained by using $\trianglelefteq$ on characteristic trees as described in [25]. While this still does not inspect the regular type information directly, it indirectly examines the influence of the regular types on the characteristic trees. We have used this approach in the experiments in the appendix.

Finally, a (relatively minor) rewrite of ECCE itself was required. In fact, we had to extend ECCE to be able to handle two kinds of predicates: ordinary predicates for which code has to be generated and constraint predicates (such as *rul_constraint*/1) for which no code has to be generated but which can be duplicated or removed "at will".

First Experiments A simple specialisation example, which illustrates the improved capabilities of our approach, can be found in Appendix B for referees. A missing feature prevented us from performing extensive benchmarks and from fully handling the CSP example in this paper (however, once corrected, we hope that our approach will be able to automatically deal with this example). This will be remedied in the full version of the paper.

6 Discussion

On the side of partial deduction, the closest work related to ours is the independently developed [13]. However, in contrast to [13] we can handle conjunctions and now have a fully functional automatic system with which we can conduct experiments (no experiments are reported in [13] although a system does probably exist by now), and which can make use of existing control techniques such as characteristic trees as described in [25]. On the side of verification, there are many related works which use logic programming based approaches for model checking [32, 5], [7], [9], [26, 24] [30] [8]. All of these works share the view that logic programming is a convenient formalism for expressing specifications and concurrent systems, and that using latest generation Prolog technology (efficient) model checking algorithms can be encoded with relatively little effort. However, to our knowledge, none of these works is capable to handle infinite state process algebras.

In future, we hope that we can prove some completeness results of our abstract partial deduction for coverability problems of well-structured process algebras (in the style of [24, 23], which proves completeness of ordinary partial deduction for coverability problems of Petri nets). Other future work is the integration of a bottom-up component. Currently only bottom-up abstract interpretation in the style of MSV [29] is implemented within ECCE; but we plan to provide support for a full RUL-based bottom-up success propagation and provide support for the fully-interleaved algorithm described in [22, 19] (a preliminary implementation of the latter algorithm already exists within the latest version of ECCE).

Acknowledgements

We would like to thank John Gallagher for his extensive support and feedback, and also for allowing us to re-use parts of his tool. We are also grateful to Fabio Fioravanti, Helko Lehmann, and Ulrich Ultes-Nitsche for stimulating discussions and feedback.

References

[1] K. R. Apt. Introduction to logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 10, pages 495–574. North-Holland Amsterdam, 1990.

[2] M. Bruynooghe. A practical framework for the abstract interpretation of logic programs. *The Journal of Logic Programming*, 10:91–124, 1991.

[3] C. Consel and S. C. Khoo. Parameterized partial evaluation. *ACM Transactions on Programming Languages and Systems*, 15(3):463–493, 1993.

[4] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *The Journal of Logic Programming*, 13(2 & 3):103–179, 1992.

[5] B. Cui, Y. Dong, X. Du, N. Kumar, C. R. Ramakrishnan, I. V. Ramakrishnan, A. Roychoudhury, S. A. Smolka, and D. S. Warren. Logic programming and model checking. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Proceedings of ALP/PLILP'98*, LNCS 1490, pages 1–20. Springer-Verlag, 1998.

[6] D. De Schreye, R. Glück, J. Jørgensen, M. Leuschel, B. Martens, and M. H. Sørensen. Conjunctive partial deduction: Foundations, control, algorithms and experiments. *The Journal of Logic Programming*, 41(2 & 3):231–277, November 1999.

[7] G. Delzanno and A. Podelski. Model checking in CLP. In R. Cleaveland, editor, *Proceedings of TACAS'99, the Second International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *LNCS*, pages 223–239. Springer-Verlag, 1999.

[8] F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying ctl properties of infinite-state systems by specializing constraint logic programs. In *Proceedings of VCL'2001*, Florence, Italy, September 2001.

[9] L. Fribourg. Constraint logic programming applied to model checking. In *Proceedings of LOPSTR'99*, LNCS 1817, pages 30–41. Springer-Verlag, 1999.

[10] J. Gallagher. Tutorial on specialisation of logic programs. In *Proceedings of PEPM'93, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–98. ACM Press, 1993.

[11] J. Gallagher. A bottom-up analysis toolkit. Technical Report CSTR-95-016, University of Bristol, July 1995.

[12] J. Gallagher and D. A. de Waal. Fast and precise regular approximations of logic programs. In P. Van Hentenryck, editor, *Proceedings of the Eleventh International Conference on Logic Programming*, pages 599–613. The MIT Press, 1994.

[13] J. P. Gallagher and J. C. Peralta. Using regular approximations for generalisation durign partial evaluation. In J. Lawall, editor, *Proceedings of PEPM'00*, pages 44–51. ACM Press, 2000. Full version to appear in Higher-Order and Symbolic Computation.

[14] M. Hermenegildo, R. Warren, and S. K. Debray. Global flow analysis as a practical compilation tool. *The Journal of Logic Programming*, 13(4):349–366, 1992.

[15] N. D. Jones. Combining abstract interpretation and partial evaluation. In P. Van Hentenryck, editor, *Static Analysis, Proceedings of SAS'97*, LNCS 1302, pages 396–405, Paris, 1997. Springer-Verlag.

[16] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[17] J.-L. Lassez, M. Maher, and K. Marriott. Unification revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan-Kaufmann, 1988.

[18] M. Leuschel. The ECCE partial deduction system and the DPPD library of benchmarks. Obtainable via `http://www.ecs.soton.ac.uk/~mal`, 1996-2000.

[19] M. Leuschel. Program specialisation and abstract interpretation reconciled. In J. Jaffar, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming JICSLP'98*, pages 220–234, Manchester, UK, June 1998. MIT Press.

[20] M. Leuschel. Logic program specialisation. In J. Hatcliff, T. Æ. Mogensen, and P. Thiemann, editors, *Partial Evaluation: Practice and Theory*, LNCS 1706, pages 155–188, Copenhagen, Denmark, 1999. Springer-Verlag.

[21] M. Leuschel. Logic program specialisation and top-down abstract interpretation reconciled. Technical Report DSSE-TR-2000-3, Department of Electronics and Computer Science, University of Southampton, May 2000.

[22] M. Leuschel and D. De Schreye. Logic program specialisation: How to be more specific. In H. Kuchen and S. Swierstra, editors, *Proceedings of the International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'96)*, LNCS 1140, pages 137–151, Aachen, Germany, September 1996. Springer-Verlag.

[23] M. Leuschel and H. Lehmann. Coverability of reset Petri nets and other well-structured transition systems by partial deduction. In J. Lloyd, editor, *Proceedings of the International Conference on Computational Logic (CL'2000)*, LNAI 1861, pages 101–115, London, UK, 2000. Springer-Verlag.

[24] M. Leuschel and H. Lehmann. Solving coverability problems of Petri nets by partial deduction. In M. Gabbrielli and F. Pfenning, editors, *Proceedings of PPDP'2000*, pages 268–279, Montreal, Canada, 2000. ACM Press.

[25] M. Leuschel, B. Martens, and D. De Schreye. Controlling generalisation and polyvariance in partial deduction of normal logic programs. *ACM Transactions on Programming Languages and Systems*, 20(1):208–258, January 1998.

[26] M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialisation. In A. Bossi, editor, Logic-Based Program Synthesis and Transformation. *Proceedings of LOPSTR'99*, LNCS 1817, pages 63–82, Venice, Italy, September 1999.

[27] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.

[28] J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11(3& 4):217–242, 1991.

[29] K. Marriott, L. Naish, and J.-L. Lassez. Most specific logic programs. *Annals of Mathematics and Artificial Intelligence*, 1:303–338, 1990.

[30] U. Nilsson and J. Lübcke. Constraint logic programming for local and symbolic model checking. In J. Lloyd, editor, *Proceedings of the International Conference on Computational Logic (CL'2000)*, LNAI 1861, pages 384–398, London, UK, 2000. Springer-Verlag.

[31] G. Puebla, J. Gallagher, and M. Hermenegildo. Towards integrating partial evaluation in a specialization framework based on generic abstract interpretation. In M. Leuschel, editor, *Proceedings of the ILPS'97 Workshop on Specialisation of Declarative Programs and its Application*, K.U. Leuven, Tech. Rep. CW 255, pages 29–38, Port Jefferson, USA, October 1997.

[32] Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In O. Grumberg, editor, *Proceedings of the International Conference on Computer-Aided Verification (CAV'97)*, LNCS 1254, pages 143–154. Springer-Verlag, 1997.

[33] A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice-Hall, 1999.

[34] E. Yardeni and E. Shapiro. A type system for logic programs. *The Journal of Logic Programming*, 10(2):125–154, 1990.

A The ECCE Procedure

Apart from the missing treatment of conjunctions [6] the following is basically the procedure implemented in the ECCE system [18] which we will employ later on.

Procedure A.1 (*generic partial deduction procedure*)

Input: a program P and a goal $\leftarrow A$

Output: a set of atoms or conjunctions $\mathcal{A}$ and a global tree γ

Initialisation: $\gamma :=$ a "global" tree with a single unmarked node, labelled by A

repeat
  pick an unmarked leaf node L in γ
  if $covered(L, \gamma)$ **then** mark L as processed
  else
    $W = whistle(L, \gamma)$

if $W \neq fail$ **then** $label(L) := generalise(L, W, \gamma)$[4]
else
mark L as processed
for all atoms $A \in leaves(U(P, label(L)))$ **do**
add a new unmarked child C of L to γ
$label(C) := A$
until all nodes are processed
output $\mathcal{A} := \{label(A) \mid A \in \gamma\}$ and γ

The procedure is parametrised by an unfolding rule U, a predicate $covered(L, \gamma)$, a whistle function $whistle(L, \gamma)$ and an generalisation function $generalise(L, W, \gamma)$.

Intuitively, $covered(L, \gamma)$ is a way of checking whether L or a generalisation of L has already been treated in the global tree γ. Formally, $covered(L, \gamma) = true$ must imply that $\exists M \in \gamma$ such that M is processed or generalised and for some substitution θ: $label(M)\theta = label(L)$. A particular implementation can decide to be more demanding and, e.g., return true only if there is another node in γ labelled by a variant of L.

The other two parameters are used to ensure termination. Intuitively, the $whistle(L, \gamma)$ is used to detect whether the branch of γ ending in L is "dangerous", in which case it returns a value different from $fail$ (i.e., it "blows"). This value should be an ancestor W of L compared to which L looked dangerous (e.g., L is bigger than W in some sense). The generalisation operation will then compute a generalisation of L and W, less likely to lead to non-termination. Formally, $generalise(L, W, \gamma)$ must be an atom which is more general than both L and W. This generalisation will replace the label of W in the global tree γ.

B A simple example using ECCE

Take the following simple input program, which is intended to simulate one particular problem that can happen during model checking of infinite state process algebras. Here, the predicate `v/1` generates ever large expressions, while `v2/1` deconstructs these expressions to see whether they contain the constant `a`.

```
v(a).
v(X) :- v2(X).
v(X) :- v(f(X)).

v2(a).
v2(f(X)) :- v2(X).

test7 :- v(b). /* fails */
```

Classical partial deduction is unable to infer failure of `test7`. To ensure termination, partial deduction will sooner or later have to abstract calls to

[4]Alternatively one could remove all descendants of W and change the label of W. This is controlled by the *parent_abstraction* switch in ECCE.

v/1. At that point, the *msg* of two (or more) calls will have to be computed. For example, we might have to compute *msg*(v(f(f(b))),v(f(f(f(f(b)))))) which is v(f(f(X))). Now, the problem is that there is no restriction upon the variable X and hence the call v(f(f(X))) can succeed (because v2(f(f(X))) can succeed, e.g., with c.a.s. X=a). In the RUL domain we can retain much more precise information about the calls to v/1. E.g., in the example below ECCE was able to fully automatically infer that the arguments to v/1 will always have the type t22 defined by:

```
t22(f(C)) :- t23(C).
t23(b).
t23(f(D)) :- t23(D).
```

This is the residual program generated by ECCE (where all post-processing optimisations have been turned off), with comments indicating how renaming has been performed. This program contains no facts and a simple bottom-up propagation (e.g., the one implemented in ECCE based upon [29]) can infer that test7__1 fails.

```
/* test7__1 --> [test7] */
test7__1 :- v__2(f(b)).

/* v__2(A) -->
[v(A),rul__constraint__declaration([t22(A)],
 [proc(y_y_t2/1,[(y_y_t2(b):-true)]),
  proc(y_t17/1,[(y_t17(b):-true),(y_t17(f(B)):-y_y_t2(B))]),
  proc(t22/1,[(t22(f(C)):-t23(C))]),
  proc(t23/1,[(t23(b):-true),(t23(f(D)):-t23(D))])])]   */
v__2(f(A)) :- v2__3(A).
v__2(A) :- v__2(f(A)).

/* v2__3(A) -->
[v2(A),rul__constraint__declaration([t23(A)],
 [proc(y_y_t2/1,[(y_y_t2(b):-true)]),
  proc(y_t17/1,[(y_t17(b):-true),(y_t17(f(B)):-y_y_t2(B))]),
  proc(t22/1,[(t22(f(C)):-t23(C))]),
  proc(t23/1,[(t23(b):-true),(t23(f(D)):-t23(D))])])]   */
v2__3(f(A)) :- v2__3(A).
```

C Another example using ECCE

The following example illustrates another potential application of our technique: removing explicit type checks from programs.

```
rev_tc(X,Res) :- rev(X,[],Res), list(Res).

rev([],A,A).
rev([H|X],A,R) :- rev(X,[H|A],R).

list(a) :- print(not_list_error(a)),nl.
list([]).
list([H|T]) :- list(T).
```

In the above program the predicate `rev_tc` performs a type check on the result. This type check should never print an error message. Unfortunately, this cannot be detected by classical conjunctive partial deduction. The technical reason is that the above program uses an accumulator (second argument of `rev/3`) which is only transferred to the output (third argument of `rev/3`) at the end of the computation. However, as the following shows, our new technique can automatically detect this. As you can see, ECCE has inferred that the `list/1` predicate is always called with with type information `t30` for its argument, where `t30` is defined by:

```
t30([]).        t30([B|C]) :- any(B),t30(C).
```

This enables ECCE to detect that the first clause of `list` is never applicable. Future versions of ECCE will also be able to handle more realistic type checks which use built-ins such as `\=/2`, as well as sometimes completely remove the type check by using a more powerful abstract unfolding.

```
/* Specialised program generated by Ecce 1.1 */
/* PD Goal: rev_tc(A,B) */
/* Parameters: Abs:p InstCheck:d Msv:w NgSlv:g Part:y Prun:n Sel:t Whstl:e
Raf:noFar:no Dce:no Poly:n Dpu:no ParAbs:yes Msvp:no Rrc:no */
/* Transformation time: 1800 ms */

rev_tc(A,B) :- rev_tc__1(A,B).

/* rev_tc__1(A,B) --> [rev_tc(A,B)] */
rev_tc__1([],[]).
rev_tc__1([A|B],C) :- rev_conj__2(B,[A],C).

/* rev_conj__2(A,B,C) -->
[rev(A,B,C),list(C),rul__constraint__declaration([any(A),ub_t29(B),any(C)],
[proc(ub_t29/1,[(ub_t29([D|E]):-any(D),t30(E))]),
 proc(t30/1,[(t30([]):-true),(t30([F|G]):-any(F),t30(G))]),
 proc(any/1,[(any(H):-true)])])]
*/
rev_conj__2([],[A|B],[A|B]) :- list__3(B).
rev_conj__2([A|B],C,D) :- rev_conj__2(B,[A|C],D).

/* list__3(A) -->
[list(A),rul__constraint__declaration([t30(A)],
[proc(t30/1,[(t30([]):-true),(t30([B|C]):-any(B),t30(C))]),
 proc(any/1,[(any(D):-true)])])]
*/
list__3([]).
list__3([A|B]) :- list__3(B).
```