The ECCE Partial Deduction System

Michael Leuschel
K.U. Leuven, Department of Computer Science
Celestijnenlaan 200 A, B-3001 Heverlee, Belgium
e-mail: michael@cs.kuleuven.ac.be
www : **http://www.cs.kuleuven.ac.be/~michael**

1 Introduction

Program specialisation, also called *partial evaluation* or *partial deduction*, is an automatic technique for program optimisation. The central idea is to specialise a given source program for a particular application domain. Program specialisation encompasses traditional compiler optimisation techniques, such as *constant folding* and *in-lining*, but uses more aggressive transformations, yielding both the possibility of obtaining (much) greater speedups and more difficulty in controlling the transformation process.

In addition to achieving important speedups, program specialisation allows the user to conceive more generally applicable programs using a more secure, readable and maintainable style. The program specialiser then takes care of transforming this general purpose, readable, but inefficient program into an efficient one. Another interesting application consists of using a program specialiser to obtain efficient implementations in the context of rapid prototyping.

Declarative programming languages, are high-level programming languages in which one only has to state *what* is to be computed and not necessarily *how* it is to be computed. Because of their clear (and often simple) semantical foundations, declarative languages offer significant advantages for the design of semantics based program analysers, transformers and optimisers. In this paper we[1] will present the program specialisation system ECCE which exploits some of these advantages in the context of logic programming.

This paper is structured as follows. A rough introduction to partial deduction is given in Section 2. In Section 3 we discuss general control issues of partial deduction while in Section 4 we discuss how termination of ECCE is ensured. In Section 5 we give an overview of some other features of ECCE. We conclude with some potential application areas of ECCE in Section 6 and a discussion in Section 7.

2 Partial Deduction

In contrast to ordinary (full) evaluation, a *partial evaluator* receives a program P along with only *part* of its input, called the *static input*. The remaining part of the input, called the *dynamic input*, will only be known at some later point in time. Given the static input S, the partial evaluator then produces a *specialised* version P_S of P which, when given the dynamic

[1] The word "we" should not be interpreted as to some royal ambitions of mine. In fact, as most chapters of this thesis are adapted from papers which other persons have co-authored, it just avoids confusing switches of narrative.

input D, produces the same output as the original program P. The goal is to exploit the static input in order to derive a more efficient program.

Partial evaluation has received considerable attention in logic programming [10, 21, 44] and functional programming (see e.g. [17] and references therein). In the context of logic programming, full input to a program P consists of a goal G and evaluation corresponds to constructing a complete SLDNF-tree for $P \cup \{G\}$. The static input then takes the form of a *partially instantiated* goal G' (and the specialised program should be correct and more efficient for all goals which are instances of G'). In contrast to other programming languages and paradigms, one can still execute P for G' and (try to) construct a SLDNF-tree for $P \cup \{G'\}$. However, because G' is not yet fully instantiated, the SLDNF-tree for $P \cup \{G'\}$ is usually infinite and ordinary evaluation will not terminate. A more refined approach to partial evaluation of logic programs is therefore required.

A technique which solves this problem is known under the name of *partial deduction*. Partial deduction originates from [20] (introductions can be found in [21, 10, 6, 26]). Its general idea is to construct a finite set of atoms $\mathcal{A}$ and a finite set of finite, but possibly *incomplete* SLDNF-trees (one for every[2] atom in $\mathcal{A}$) which "cover" the possibly infinite SLDNF-tree for $P \cup \{G'\}$. The derivation steps in these SLDNF-trees correspond to the computation steps which have been performed beforehand by the *partial deducer* and the clauses of the specialised program are then extracted from these trees by constructing one specialised clause per branch.

In [36], Lloyd and Shepherdson presented a fundamental correctness theorem for partial deduction. The two basic requirements for correctness of a partial deduction of P wrt $\mathcal{A}$ are the *independence* and *closedness* conditions. The independence condition requires that no two atoms in $\mathcal{A}$ have a common instance, guaranteeing that the specialised program does not produce additional answers. The closedness condition requires that all atoms in the leaves of the SLDNF-trees are instances of an atom in $\mathcal{A}$. This guarantees that all calls, which might occur during the execution of the specialised program, are covered by some specialised predicate definition.

3 Control of Partial Deduction

On-line vs. Off-line Control

The control problems of partial evaluation and deduction in general have been tackled from two different angles: the so-called *off-line* versus *on-line* approaches. The *on-line* approach performs all the control decisions *during* the actual specialisation phase. It is within this methodology that the partial deduction system ECCE is situated. The *off-line* approach on the other hand performs an analysis phase *prior* to the actual specialisation phase, based on some rough descriptions of what kinds of specialisations will have to be performed. The analysis phase provides annotations which then guide the control aspect of the proper specialisation phase, often to the point of making it completely trivial.

Partial evaluation of functional programs [5, 17] has mainly stressed off-line approaches, while supercompilation of functional [52, 51, 14] and partial deduction of logic programs [11, 50, 1, 2, 40, 42, 24, 32] have concentrated on on-line control.

The main reason for using the off-line approach is to make specialisation more amenable to effective self-application [18]. On-line methods, however, usually obtain better specialisation,

[2] Formally, an SLDNF-tree is obtained from an atom or goal by what is called an ***unfolding rule***.

because no control decisions have to be taken beforehand, i.e. at a point where the full specialisation information is not yet available.

Global vs. Local Control

In partial deduction one usually distinguishes two levels of control [10, 42]:

- the *global control*, in which one chooses the set $\mathcal{A}$, i.e. one decides *which* atoms will be partially deduced, and
- the *local control*, in which one constructs the finite (possibly incomplete) SLDNF-trees for each individual atom in $\mathcal{A}$ and thus determines *what* the definitions for the partially deduced atoms look like.

The control of partial deduction should ensure termination (see Section 4 below), adequate specialisation and correctness (i.e. ensuring the independence and closedness conditions).

In practice the independence condition is usually ensured by a *renaming*[3] transformation, and the control of partial deduction does not have to worry about this aspect of correctness.

The closedness condition can simply be ensured by repeatedly adding the uncovered leaf atoms to $\mathcal{A}$ and unfolding them. Unfortunately this process generally leads to non-termination. For instance, the "reverse with accumulating parameter" program (see e.g. [39, 41]) exposes this non-terminating behaviour. Thus, to ensure finiteness of $\mathcal{A}$, one usually applies an abstraction operator.

Definition 3.1 (abstraction) Let $\mathcal{A}$ and $\mathcal{A}'$ be sets of atoms. Then $\mathcal{A}'$ is an *abstraction* of $\mathcal{A}$ iff every atom in $\mathcal{A}$ is an instance of an atom in $\mathcal{A}'$. An *abstraction operator* is an operator which maps every finite set of atoms to a finite abstraction of it.

The abstraction operator thus decides upon the final set $\mathcal{A}$ and can therefore be seen as encapsulating the global control of partial deduction. Similarly, the local control component is usually encapsulated in what is called an unfolding rule, defined as follows.

Definition 3.2 (unfolding rule) An *unfolding rule* U is a function which, given a program P and a goal G, returns a finite and possibly incomplete SLDNF-tree for $P \cup \{G\}$.

4 Ensuring Termination

The aspect of termination can also be divided into a local and a global one. First, the problem of keeping each SLDNF-tree finite is referred to as the *local* termination problem. Secondly keeping the set $\mathcal{A}$ finite is referred to as the *global* termination problem.

To ensure termination *well-founded orders* can be used [2, 41, 40, 39]. However, in an on-line setting, well-founded orders are sometimes too rigid or too complex [27]. The ECCE partial deduction system therefore mainly relies on *well-quasi orders* to ensure local and global termination (although other unfolding rules and abstraction operators can be added to the system by the advanced user).

Well-quasi orders are formally defined as follows.

[3] Renaming maps dependent atoms to new predicate symbols and thus generates an independent set without precision loss. For instance, the dependent set $\mathcal{A} = \{member(a, L), member(X, [b])\}$ can be transformed into the independent set $\mathcal{A}' = \{member(a, L), member'(X, [b])\}$.

Definition 4.1 A quasi order $\leq_V$ (i.e. a transitive and reflexive binary relation) on a set V is called a *well-quasi-order (wqo)* on V iff for any infinite sequence of elements $e_1, e_2, \ldots$ in V there are $i < j$ such that $e_i \leq_V e_j$.

An interesting wqo is the homeomorphic embedding relation $\trianglelefteq$, which derives from results by Higman [16] and Kruskal [22]. It has been used in the context of term rewriting systems in [8], and adapted for use in supercompilation ([52]) in [51]. Its usefulness as a stop criterion for partial evaluation is also discussed and advocated in [37]. Some complexity results are summarised in[37]

The following is the definition from [51], which adapts the pure homeomorphic embedding from [8] by adding a rudimentary treatment of variables.

Definition 4.2 ($\trianglelefteq$) The *(pure) homeomorphic embedding* relation $\trianglelefteq$ on expressions is defined inductively as follows:

1. $X \trianglelefteq Y$ for all variables X, Y
2. $s \trianglelefteq f(t_1, \ldots, t_n)$ if $s \trianglelefteq t_i$ for some i
3. $f(s_1, \ldots, s_n) \trianglelefteq f(t_1, \ldots, t_n)$ if $\forall i \in \{1, \ldots, n\} : s_i \trianglelefteq t_i$.

The intuition behind the above definition is that $A \trianglelefteq B$ iff A can be obtained from B by "striking out" certain parts, or said another way, the structure of A reappears within B. For instance we have $p(a) \trianglelefteq p(f(a))$ and indeed $p(a)$ can be obtained from $p(f(a))$ by "striking out" the f. Observe that the "striking out" corresponds to the application of the diving rule 2. We also have e.g. that: $X \trianglelefteq X$, $p(X) \trianglelefteq p(f(Y))$, $p(X, X) \trianglelefteq p(X, Y)$ and $p(X, Y) \trianglelefteq p(X, X)$.

While $\trianglelefteq$ is very generous and has a lot of desirable properties it still suffers from some drawbacks. Indeed, as can be observed in the above example, the homeomorphic embedding relation $\trianglelefteq$ as defined in Definition 4.2 is rather crude wrt variables. In fact, all variables are treated as if they were the same variable, a practice which is clearly undesirable in a logic programming context. Intuitively, in the above example, $p(X, Y) \trianglelefteq p(X, X)$ is acceptable, while $p(X, X) \trianglelefteq p(X, Y)$ is not. Indeed $p(X, X)$ can be seen as standing for something like $and(eq(X, Y), p(X, Y))$, which clearly embeds $p(X, Y)$, but the reverse does not hold.

Secondly, $\trianglelefteq$ behaves in quite unexpected ways in the context of generalisation, where it can pose some subtle problems wrt the termination of a generalisation process (see [27]).

To remedy these problems, [32, 26, 33] introduced the so called strict homeomorphic embedding, which is used by the ECCE system:

Definition 4.3 ($\trianglelefteq^*$) Let A, B be expressions. Then B *(strictly homeomorphically) embeds* A, written as $A \trianglelefteq^* B$, iff $A \trianglelefteq B$ and A is not a strict instance of B.

We now still have that $p(X, Y) \trianglelefteq^* p(X, X)$ but not $p(X, X) \trianglelefteq^* p(X, Y)$. Note that still $X \trianglelefteq^* Y$ and $X \trianglelefteq^* X$.

5 Other Features of the System

In this section we present some other aspects of the ECCE system.

5.1 Global Control: Characteristic Trees and Atoms

The global control is often considered to be the more difficult part of controlling partial deduction. A refined approach is the one based on *characteristic trees* [11, 9], which capture the *specialisation behaviour* of atoms. An abstraction operator like the most specific generalisation[4] (*msg*) is just based on the *syntactic structure* of the atoms to be specialised. This is generally not such a good idea. Indeed, two atoms can be specialised very similarly in the context of one program P_1 and very dissimilarly in the context of another one P_2. Characteristic trees [11, 9], however, capture (to some depth) how the atoms are specialised and how they behave computationally in the context of the respective programs. An abstraction operator which takes these trees into account will notice their similar behaviour in the context of P_1 and their dissimilar behaviour within P_2, and can therefore take appropriate actions in the form of different generalisations.

Unfortunately, as shown in [30], it is in general impossible to preserve characteristic trees upon generalisation in the context of ordinary partial deduction. This can lead to severe specialisation losses, as well as to non-termination of certain partial deduction algorithms.

A solution to this entanglement is presented in [24, 26], where it is called *ecological partial deduction*. Its basic idea is to simply *impose* characteristic trees on the generalised atoms. This amounts to associating characteristic trees with the atoms to be specialised — called *characteristic atoms* — allowing the preservation of characteristic trees in a straightforward way and circumventing the need for intricate generalisations. This feature is available in the ECCE system.

However, ecological partial deduction still requires a depth bound on the characteristic trees in order to ensure termination. In [32, 26] it is shown that this leads to undesired results in cases where the depth bound is actually required. A solution to this problem can be obtained by combining ecological partial deduction with the (global) m-trees of [42]: The basic idea is to structure combinations of atoms and associated characteristic trees in a global tree, registering "causal" relationships among such pairs. By adapting the homeomorphic embedding relation for characteristic trees one can then spot potential sequences of ever growing characteristic trees and perform proper generalisation in order to avert the danger (without having to impose a depth bound on characteristic trees). The details of this approach have been elaborated in [32, 26] and it is implemented in the ECCE system.

5.2 Conjunctive Partial Deduction

Conjunctive partial deduction has been designed with the aim of overcoming some limitations inherent in "classical" partial deduction. The essential aspect lies in the joint treatment of *entire conjunctions* of atoms, connected through shared variables, at the global level (complemented with some renaming to deliver program clauses). Basically, this can be seen as a refinement of abstraction with respect to the conventional case. Indeed, in conjunctive partial deduction, a conjunction can be abstracted by either splitting it into subconjunctions, or generalising syntactic structure, or through a combination of both. In classical partial deduction, on the other hand, any conjunction is always split (i.e. abstracted) into its constituent atoms before lifting the latter to the global level. Details can be found in [31, 13, 26].

Apart from this aspect, the conventional control notions described above also apply in a conjunctive setting. Notably, the concept of characteristic atoms can be generalised to

[4] Also known as anti-unification or least general generalisation, see e.g. [23].

characteristic conjunctions, which are just pairs consisting of a conjunction and an associated characteristic tree.

Conjunctive partial deduction enables the ECCE system to e.g. perform *tupling* and *deforestation* (see Section 6), but it also diminishes the need for aggressive unfolding rules and allows to reconcile specialisation and efficiency of the specialised program (see [31, 26]).

5.3 Treatment of Built-ins

The ECCE system also handles a lot of Prolog built-ins, like for instance =, is, <, =<, <, >=, *number*, *atomic*, *call*, \==, \=. All built-ins are supposed to be declarative and their selection delayed until they are sufficiently instantiated. Thus ECCE falls more into the line of the SP [9, 10] system: both, unlike MIXTUS [50, 49] and PADDY [45, 46, 47], do not attempt to handle *full* Prolog.

The global control technique sketched in Section 5.1 is extended by also registering built-ins in the characteristic trees (See [26].) Also, in order to adequately handle some built-ins, the embedding relation $\trianglelefteq$ of Definition 4.2 has to be adapted. Indeed, some built-ins (like $= ../2$ or $is/2$) can be used to dynamically construct infinitely many new constants and functors and thus $\trianglelefteq$ is no longer a wqo.

To remedy this, the constants and functors are partitioned into the *static* ones, occurring in the original program and the partial deduction query, and the *dynamic* ones. (This approach is also used in [49, 50].) The set of dynamic constants and functors is possibly infinite, and are therefore treated like the infinite set of variables in Definition 4.2 by adding the following rule to the ECCE system:

$f(s_1, \ldots, s_m) \trianglelefteq^* g(t_1, \ldots, t_n)$ if both f and g are dynamic

More refined approaches are possible [27], but are not yet implemented in the current version of ECCE.

5.4 More Specific Resolution Steps

Another feature implemented in ECCE is the *more specific resolution* step, first presented in [9]. The basic idea of the technique is to instantiate a goal by tentatively unfolding it and then taking the *msg* of all so obtained instantiations. This achieves a simple kind of *side-ways information passing* (see, however, the discussions in [29]). Take e.g. the following program:

```
rotate(leaf(N),leaf(N)).
rotate(tree(L,N,R),tree(LR,N,RR)) :- rotate(L,LR),rotate(R,RR).
rotate(tree(L,N,R),tree(RR,N,LR)) :- rotate(L,LR),rotate(R,RR).
prune(leaf(N),leaf(N)).
prune(tree(L,0,R),leaf(0)).
prune(tree(L,s(N),R),tree(LP,s(N),RP)) :- prune(L,LP),prune(R,RP).
```

as well as the following goal:

```
?- rotate(tree(L,0,R),TR),prune(TR,TRP).
```

Neither atom in the goal is determinate (without lookahead). After a more specific resolution step which tentatively unfolds rotate(tree(L,0,R),TR) we obtain:

```
?- rotate(tree(L,0,R),tree(L1,0,R1)),prune(tree(L1,0,R1),TRP).
```

A simple side-ways information passing has been accomplished, which in this case reduces the search space and improves the detection of determinacy (prune(tree(L1,0,R1),TRP) matches only 1 clause while prune(TR,TRP) matched all 3 clauses of prune).

5.5 Postprocessing

Redundant Argument Filtering (RAF and FAR)

RAF is a technique developed in [35, 26] which detects certain[5] redundant arguments. Basically, it detects those arguments which are existential and which can thus be safely removed.

RAF is very useful when performed after conjunctive partial deduction. On a range of programs produced by conjunctive partial deduction (with renaming) in [35, 26], RAF reduced code size and execution time by an average of approximately 20%. The algorithm never increased code size nor execution time.

But redundant arguments also arise when one re-uses generic predicates for more specific purposes. For instance, let us define a *member/2* predicate by re-using a generic *delete/3* predicate:

```
member(X,L) :- delete(X,L,DL).
delete(X,[X|T],T).
delete(X,[Y|T],[Y|DT]) :- delete(X,T,DT).
```

Here the third argument of *delete* is redundant and will be removed by the ECCE system if RAF is enabled:

```
member(X,L) :- delete(X,L).
delete(X,[X|T]).
delete(X,[Y|T]) :- delete(X,T).
```

The ECCE system also contains the reverse argument filtering (FAR) of [35, 26] ("reverse" because the safety conditions are reversed wrt RAF). While RAF detects existential arguments (which might return a computed answer binding), FAR detects arguments which can be non-existential and non-ground but whose value is never used (and for which no computed answer binding will be returned). Take e.g. the following program:

```
p(X) :-  q(f(X)).
q(Z).
```

Here the argument of $q(f(X))$ is not a variable but the value is never used. The ECCE system will remove this argument if FAR is enabled:

```
p(X) :-  q.
q.
```

Determinate Postunfolding (DPU) and Dead Code Elimination (DCE)

The determinate postunfolding phase searches for atoms in the body of clauses which only match a single clause. These atoms will then be unfolded (given some safety criterion to ensure termination). This simple postprocessing phase can be highly beneficial, but also often gives rise to dead code, i.e. code that is unreachable (in the predicate dependency

[5] Finding all redundant arguments is undecidable in general [35, 26].

graph) from the goal G for which a program is specialised. The dead code elimination phase therefore takes care to remove such code.

While generating code, ECCE will also remove useless clauses of the form $H \leftarrow F, H, G$.

6 Some Applications of the ECCE system

Below we present some possible applications of the ECCE system. Unless explicitly stated otherwise, all experiments were conducted with the default settings of ECCE. For other examples, the interested reader may consult [25] and also [26].

6.1 Removing Meta-Interpretation Overhead

Removing interpretation overhead is one of the traditional (and successful) applications of partial evaluation. The following is an interpreter for the ground representation (this particular implementation is called the mixed representation in [28, 26]), slightly adapted from the one already specialised in [9].

```
solve(GrRules,[]).
solve(GrRules,[NgH|NgT]) :-
    non_ground_member(term(clause,[NgH|NgBody]),GrRules),
    solve(GrRules,NgBody),solve(GrRules,NgT).

non_ground_member(NgX,[GrH|GrT]) :- make_non_ground(GrH,NgX).
non_ground_member(NgX,[GrH|GrT]) :- non_ground_member(NgX,GrT).

make_non_ground(G,NG) :- mkng(G,NG,[],Sub).

mkng(var(N),X,[],[sub(N,X)]).
mkng(var(N),X,[sub(N,X)|T],[sub(N,X)|T]).
mkng(var(N),X,[sub(M,Y)|T],[sub(M,Y)|T1]) :- N \= M,mkng(var(N),X,T,T1).
mkng(term(F,Args),term(F,IArgs),InSub,OutSub) :- l_mkng(Args,IArgs,InSub,OutSub).

l_mkng([],[],Sub,Sub).
l_mkng([H|T],[IH|IT],InSub,OutSub) :-
    mkng(H,IH,InSub,IntSub),l_mkng(T,IT,IntSub,OutSub).
```

Specialisation of this meta-interpreter for the following query, i.e. for append as the object-level program together with some partial input at the object-level,

```
?-solve([term(clause,[term(app,[term(null,[]),var(l),var(l)])]),
         term(clause,[term(app,[term(cons,[var(h),var(x)]),var(y),
           term(cons,[var(h),var(z)])]),term(app,[var(x),var(y),var(z)])])],
        [term(app,[term(cons,[term(a,[]),term(cons,[term(b,[]),X])]),Y,Z])]).
```

results in the following specialised program in which almost all of the overhead of the ground representation has been removed and in which even specialisation at the object-level has been performed (something which off-line systems cannot do, in one go at least):

```
 solve__1(X1,X2,term(cons,[term(a,[]),term(cons,[term(b,[]),X3])])) :-
    non_ground_member_conj__3(X1,X2,X3).
non_ground_member_conj__3(term(null,[]),X1,X1).
non_ground_member_conj__3(term(cons,[X1,X2]),X3,term(cons,[X1,X4])) :-
    non_ground_member_conj__3(X2,X3,X4).
```

6.2 Removing Higher-Order Overhead

One can often get rid of the overhead of higher-order programming via partial deduction in general (see also [43] using MIXTUS) and ECCE in particular. Take the following program containing implementations of the higher-order predicates map and reduce.

```
map(P,[],[]).
map(P,[H|T],[PH|PT]) :- Call =.. [P,H,PH],call(Call),map(P,T,PT).

reduce(Func,Base,[],Base).
reduce(Func,Base,[H|T],Res) :-
     reduce(Func,Base,T,TRes),Call =.. [Func,H,TRes,Res],call(Call).

reduce_add(List,Res) :- reduce(add,0,List,Res).
add(X,Y,Z) :- Z is X + Y.
```

By specialising this program with ECCE for the goal map(reduce_add,X,Y) the overhead of the higher-order programming has been removed:[6]

```
map(reduce_add,X1,X2) :- map__1(X1,X2).
map__1([],[]).
map__1([X1|X2],[X3|X4]) :- reduce_add__2(X1,X3),map__1(X2,X4).

reduce_add__2([],0).
reduce_add__2([X1|X2],X3) :- reduce__3(X2,X4),X3 is '+'(X1,X4).
reduce__3([],0).
reduce__3([X1|X2],X3) :- reduce__3(X2,X4),X3 is '+'(X1,X4).
```

6.3 Tupling and Deforestation

Conjunctive partial deduction allows the ECCE system to perform *tupling* (translating multiple visits of the same data structure into a single visit, see e.g. [44]) and, together with redundant argument filtering explained in Section 5.5 above, *deforestation* (getting rid of intermediate data structures, see [55]). These optimisations allow the composition of different simple predicates — performing just one task and which are therefore simpler to write and verify — without incurring the resulting overhead.

Take for example the following program, in which a double-append predicate is written in a straightforward manner by reusing the normal append:

```
da(X,Y,Z,R) :- app(X,Y,I),app(I,Z,R).
app([],L,L).
app([H|X],Y,[H|Z]) :- app(X,Y,Z).
```

The predicate da is simple to write but quite inefficient because an intermediate list I is unnecessarily constructed and re-visited. By invoking ECCE for the query da(X,Y,Z,R) one obtains the following specialised program:

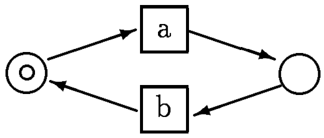
```
da([],Y,Z,R) :- app(Y,Z,R).
da([H|X],Y,Z,[H|R]) :- da(X,Y,Z,R).
app([],L,L).
app([H|X],Y,[H|Z]) :- app(X,Y,Z).
```

[6] Using a more sophisticated post-processing, it would be possible to merge the two predicates reduce__2 and reduce__3. Such an optimisation is planned for a future release of ECCE.

which no longer constructs or traverses the intermediate list I. See [19, 26] for further details and for current limitations of ECCE concerning tupling and deforestation (getting rid of these limitations is a topic of further research).

6.4 Reachability Analysis

Using partial deduction it is possible to analyse the reachable states of logic programs (or other constructs mapped to logic programs via meta-programming). As a simple example, we show how we can use ECCE to analyse reachable markings or safeness properties of Petri Nets [48]. Take for instance the following simple Petri net:



By a straightforward translation one obtains a (non-terminating) logic program which calculates the reachable markings [48] of the Petri net:

```
state(X,Y,reach(X,Y)).
state(s(X),Y,R) :- state(X,s(Y),R).  /* transition a */
state(X,s(Y),R) :- state(s(X),Y,R).  /* transition b */
```

The first two arguments of state represent the current marking (i.e. the number of tokens in each place) while the third argument is the output.

When specialising this program for the query state(s(0),0,X1) we obtain the following specialised program (for the examples in this subsection we used a slightly more aggressive unfolding rule than the one selected in the default setting):

```
state__1(reach(s(0),0)).
state__1(reach(0,s(0))).
```

To analyse 1-safeness [48] of the above Petri net we can use the following straightforward logic program translation:

```
unsafe(s(s(_X)),_Y).
unsafe(_X,s(s(_Y))).
unsafe(s(X),Y) :- unsafe(X,s(Y)).  /* transition a */
unsafe(X,s(Y)) :- unsafe(s(X),Y).  /* transition b */
```

When specialising unsafe(s(0),0) we get the following program, i.e. the Petri net is 1-safe for the particular initial marking in the above figure:

```
unsafe(s(0),0) :- fail.
```

When specialising unsafe(s(0),s(0)) we get the following program, i.e. for an initial marking with 1 token in each place the Petri net is not 1-safe:

```
unsafe(s(0),s(0)).
```

Observe that none of the above queries (universally) terminate under Prolog or SLD-resolution.

6.5 Inductive Theorem Proving

The relation between program specialisation and theorem proving has already been raised several times in the literature [53, 12, 54]. In this subsection we show how ECCE can be used to perform some basic inductive theorem proving. Take the following simple program.

```
even(0).
even(s(X)) :- odd(X).
odd(s(X)) :- even(X).
```

By specialising this program with ECCE, combined with a more specific program construction phase [38] (another, optional, post-processing phase implemented in ECCE), for the query even(X),odd(X) one can derive that the query fails and that no number is even and odd at the same time. This example poses difficulties for quite a lot of abstract interpretation methods.

Next we show how one can derive the associativity of the plus operation on natural numbers using the ECCE system (with default settings[7]).

```
plus(0,X,X).
plus(s(X),Y,s(Z)) :- plus(X,Y,Z).

pasoc(X,Y,Z,R1,R2) :- plus(X,Y,XY),plus(XY,Z,R1),  plus(Y,Z,YZ),plus(X,YZ,R2).
```

After specialisation and more specific program construction we get the following program, were the fact that R1 and R2 are identical (i.e. plus is associative) has been deduced (arguments 4 and 5 of pasoc are identical).

```
pasoc(X1,X2,X3,X4,X4) :- pasoc__1(X1,X2,X3,X4,X4).

pasoc__1(0,X1,X2,X3,X3) :- plus_conj__3(X1,X2,X3,X3).
pasoc__1(s(X1),X2,X3,s(X4),s(X4)) :- plus_conj__2(X1,X2,X3,X4,X4).
plus_conj__2(0,X1,X2,X3,X3) :- plus_conj__3(X1,X2,X3,X3).
plus_conj__2(s(X1),X2,X3,s(X4),s(X4)) :- plus_conj__2(X1,X2,X3,X4,X4).
plus_conj__3(0,X1,X1,X1).
plus_conj__3(s(X1),X2,s(X3),s(X3)) :- plus_conj__3(X1,X2,X3,X3).
```

In future work we plan to integrate *constrained* partial deduction [30] as well as the more refined algorithm of [29], which interleaves bottom-up abstract interpretation steps with top-down conjunctive partial deduction unfolding steps, into the ECCE system. First investigations indicate that this will make it possible to prove much more sophisticated inductive theorems as well as provide increased specialisation capabilities in other settings as well.

7 Further Discussion and Conclusion

Various experiments with the ECCE system can be found in [33, 19, 26]. The results indicate definite improvements over SP [9, 10] MIXTUS [50, 49] and PADDY [45, 46, 47] (for declarative

[7]Which is somewhat surprising, as the associativity of plus is the classical example to illustrate the rippling heuristic in inductive theorem proving (e.g. in [3]). No such heuristic is required here. Nonetheless, we believe that a fruitful cross-fertilisation between control techniques for inductive theorem proving and partial deduction is possible and desirable.

logic programs; to specialise full Prolog programs MIXTUS and PADDY remain the systems of choice).

Several areas of ECCE can be pinpointed for further refinement. Implementing constructive negation [4] as well as providing support for the if-then-else construct — both to be found e.g. in SAGE [15] — are essential for certain applications. Support for tabled logic programming, using the ideas developed in [34], will also be useful in practice.

Although, at least for the default settings, ECCE will rarely worsen the speed of a program, a good way to fully prevent slowdowns in practice remains a pressing open question. Also, in some cases (e.g. when specialising the ground representation with partially static arguments) the present global control regime will allow way too many versions as warranted by the actual specialisation that can be achieved. For both of these problems, a promising direction might be to incorporate a more detailed efficiency and cost estimation into the global and local control of (e.g. based on [7]).

The speed of the ECCE system itself — currently still a prototype in a lot of respects — is another area of possible improvement (although the transformation times are comparable to those of MIXTUS). One could try to incorporate ideas from [47] which allow the PADDY system to be very efficient. Some time should also be spent on a more efficient implementation of $\unlhd^*$ (the current implementation is still quite naive).

Availability and Pricing

The ECCE partial deduction system is freely available (for non-commercial and non-military use) at [25] together with the DPPD (Dozens of Problems for Partial Deduction) library of benchmarks. Further details about the various techniques used in the ECCE system can most conveniently be found in [26] or by consulting [24, 32, 33, 31, 13, 35, 19].

Acknowledgements

Michael Leuschel is supported by the Belgian GOA "Non-Standard Applications of Abstract Interpretation". Most parts of the ECCE system grew out of joint work with D. De Schreye, R. Glück, J. Jørgensen, B. Martens, M.H. Sørensen and A. de Waal. I am also grateful to Wim Vanhoof, the most intensive ECCE user so far. Finally, I would like to thank Germán Puebla for inviting this paper to the ILPS'97 workshop on "Tools and Environments for (Constraint) Logic Programming."

References

[1] R. Bol. Loop checking in partial deduction. *The Journal of Logic Programming*, 16(1&2):25–46, 1993.

[2] M. Bruynooghe, D. De Schreye, and B. Martens. A general criterion for avoiding infinite unfolding during partial deduction. *New Generation Computing*, 11(1):47–79, 1992.

[3] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: a heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253, 1993.

[4] D. Chan and M. Wallace. A treatment of negation during partial evaluation. In H. Abramson and M. Rogers, editors, *Meta-Programming in Logic Programming, Proceedings of the Meta88 Workshop, June 1988*, pages 299–318. MIT Press, 1989.

[5] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *Proceedings of ACM Symposium on Principles of Programming Languages (POPL'93)*, Charleston, South Carolina, January 1993. ACM Press.

[6] D. De Schreye, M. Leuschel, and B. Martens. Tutorial on program specialisation (abstract). In J. W. Lloyd, editor, *Proceedings of ILPS'95, the International Logic Programming Symposium*, Portland, USA, December 1995. MIT Press.

[7] S. Debray and N.-W. Lin. Cost analysis of logic programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, November 1993.

[8] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pages 243–320. Elsevier, MIT Press, 1990.

[9] J. Gallagher. A system for specialising logic programs. Technical Report TR-91-32, University of Bristol, November 1991.

[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 and M. Bruynooghe. The derivation of an algorithm for program specialisation. *New Generation Computing*, 9(3 & 4):305–333, 1991.

[12] R. Glück and J. Jørgensen. Generating transformers for deforestation and supercompilation. In B. Le Charlier, editor, *Proceedings of SAS'94*, LNCS 864, pages 432–448, Namur, Belgium, September 1994. Springer-Verlag.

[13] R. Glück, J. Jørgensen, B. Martens, and M. H. Sørensen. Controlling conjunctive partial deduction of definite logic programs. In H. Kuchen and S. Swierstra, editors, *Proceedings of the International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'96)*, LNCS 1140, pages 152–166, Aachen, Germany, September 1996. Springer-Verlag.

[14] R. Glück and M. H. Sørensen. A roadmap to supercompilation. In O. Danvy, R. Glück, and P. Thiemann, editors, *Proceedings of the 1996 Dagstuhl Seminar on Partial Evaluation*, LNCS 1110, pages 137–160, Schloß Dagstuhl, 1996. Springer-Verlag.

[15] C. A. Gurr. *A Self-Applicable Partial Evaluator for the Logic Programming Language Gödel*. PhD thesis, Department of Computer Science, University of Bristol, January 1994.

[16] G. Higman. Ordering by divisibility in abstract algebras. *Proceedings of the London Mathematical Society*, 2:326–336, 1952.

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

[18] N. D. Jones, P. Sestoft, and H. Søndergaard. Mix: a self-applicable partial evaluator for experiments in compiler generation. *LISP and Symbolic Computation*, 2(1):9–50, 1989.

[19] J. Jørgensen, M. Leuschel, and B. Martens. Conjunctive partial deduction in practice. In J. Gallagher, editor, *Proceedings of the International Workshop on Logic Program Synthesis and Transformation (LOPSTR'96)*, LNCS 1207, pages 59–82, Stockholm, Sweden, August 1996. Springer-Verlag. Also in the Proceedings of BENELOG'96. Extended version as Technical Report CW 242, K.U. Leuven.

[20] J. Komorowski. Partial evaluation as a means for inferencing data structures in an applicative language: a theory and implementation in the case of prolog. In *Ninth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. Albuquerque, New Mexico*, pages 255–267, 1982.

[21] J. Komorowski. An introduction to partial deduction. In A. Pettorossi, editor, *Proceedings Meta'92*, LNCS 649, pages 49–69. Springer-Verlag, 1992.

[22] J. B. Kruskal. Well-quasi ordering, the tree theorem, and Vazsonyi's conjecture. *Tansactions of the American Mathematical Society*, 95:210–225, 1960.

[23] 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.

[24] M. Leuschel. Ecological partial deduction: Preserving characteristic trees without constraints. In M. Proietti, editor, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'95*, LNCS 1048, pages 1–16, Utrecht, The Netherlands, September 1995. Springer-Verlag.

[25] M. Leuschel. The ECCE partial deduction system and the DPPD library of benchmarks. Obtainable via `http://www.cs.kuleuven.ac.be/~lpai`, 1996.

[26] M. Leuschel. *Advanced Techniques for Logic Program Specialisation*. PhD thesis, K.U. Leuven, May 1997. Accessible via `http://www.cs.kuleuven.ac.be/~michael`.

[27] M. Leuschel. Extending homeomorphic embedding in the context of logic programming. Technical Report CW 252, Departement Computerwetenschappen, K.U. Leuven, Belgium, June (revised August) 1997. To appear in the Proceedings of the 12th Workshop Logische Programmierung (WLP'97), München, Germany, September 1997. Accessible via `http://www.cs.kuleuven.ac.be/~lpai`.

[28] M. Leuschel and D. De Schreye. Towards creating specialised integrity checks through partial evaluation of meta-interpreters. In *Proceedings of PEPM'95, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 253–263, La Jolla, California, June 1995. ACM Press.

[29] 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. Extended version as Technical Report CW 232, K.U. Leuven.

[30] M. Leuschel and D. De Schreye. Constrained partial deduction and the preservation of characteristic trees. *New Generation Computing*. To appear. Preliminary version as

Technical Report CW 250, Departement Computerwetenschappen, K.U. Leuven, Belgium, June 1997. Accessible via http://www.cs.kuleuven.ac.be/~lpai.

[31] M. Leuschel, D. De Schreye, and A. de Waal. A conceptual embedding of folding into partial deduction: Towards a maximal integration. In M. Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming JICSLP'96*, pages 319–332, Bonn, Germany, September 1996. MIT Press. Extended version as Technical Report CW 225, K.U. Leuven.

[32] M. Leuschel and B. Martens. Global control for partial deduction through characteristic atoms and global trees. In O. Danvy, R. Glück, and P. Thiemann, editors, *Proceedings of the 1996 Dagstuhl Seminar on Partial Evaluation*, LNCS 1110, pages 263–283, Schloß Dagstuhl, 1996. Springer-Verlag. Extended version as Technical Report CW 220, K.U. Leuven.

[33] 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*. To appear. Preliminary version as Technical Report CW 248, Departement Computerwetenschappen, K.U. Leuven, Belgium, February 1996.

[34] M. Leuschel, B. Martens, and K. Sagonas. Preserving termination of tabled logic programs while unfolding. In N. Fuchs, editor, *Pre-Proceedings of the International Workshop on Logic Program Synthesis and Transformation (LOPSTR'97)*, Leuven, Belgium, July 1997.

[35] M. Leuschel and M. H. Sørensen. Redundant argument filtering of logic programs. In J. Gallagher, editor, *Proceedings of the International Workshop on Logic Program Synthesis and Transformation (LOPSTR'96)*, LNCS 1207, pages 83–103, Stockholm, Sweden, August 1996. Springer-Verlag. Extended version as Technical Report CW 243, K.U. Leuven.

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

[37] R. Marlet. *Vers une Formalisation de l'Évaluation Partielle*. PhD thesis, Université de Nice - Sophia Antipolis, December 1994.

[38] K. Marriott, L. Naish, and J.-L. Lassez. Most specific logic programs. *Annals of Mathematics and Artificial Intelligence*, 1:303–338, 1990. Preliminary version in *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 909–923, Seattle, 1988. IEEE, MIT Press.

[39] B. Martens. *On the Semantics of Meta-Programming and the Control of Partial Deduction in Logic Programming*. PhD thesis, K.U. Leuven, February 1994.

[40] B. Martens and D. De Schreye. Automatic finite unfolding using well-founded measures. *The Journal of Logic Programming*, 28(2):89–146, August 1996.

[41] B. Martens, D. De Schreye, and T. Horváth. Sound and complete partial deduction with unfolding based on well-founded measures. *Theoretical Computer Science*, 122(1–2):97–117, 1994.

[42] B. Martens and J. Gallagher. Ensuring global termination of partial deduction while allowing flexible polyvariance. In L. Sterling, editor, *Proceedings ICLP'95*, pages 597–613, Kanagawa, Japan, June 1995. MIT Press.

[43] L. Naish. Higher-order logic programming in Prolog. Technical Report 96/2, Department of Computer Science, University of Melbourne, 1995.

[44] A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *The Journal of Logic Programming*, 19& 20:261–320, May 1994.

[45] S. Prestwich. The PADDY partial deduction system. Technical Report ECRC-92-6, ECRC, Munich, Germany, 1992.

[46] S. Prestwich. An unfold rule for full Prolog. In K.-K. Lau and T. Clement, editors, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'92*, Workshops in Computing, University of Manchester, 1992. Springer-Verlag.

[47] S. Prestwich. Online partial deduction of large programs. In *Proceedings of PEPM'93, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 111–118. ACM Press, 1993.

[48] W. Reisig. *Petri Nets - An Introduction.* Springer Verlag, 1982.

[49] D. Sahlin. *An Automatic Partial Evaluator for Full Prolog.* PhD thesis, Swedish Institute of Computer Science, March 1991.

[50] D. Sahlin. Mixtus: An automatic partial evaluator for full Prolog. *New Generation Computing*, 12(1):7–51, 1993.

[51] M. H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In J. W. Lloyd, editor, *Proceedings of ILPS'95, the International Logic Programming Symposium*, pages 465–479, Portland, USA, December 1995. MIT Press.

[52] V. F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.

[53] V. F. Turchin. Program transformation with metasystem transitions. *Journal of Functional Programming*, 3(3):283–313, 1993.

[54] V. F. Turchin. Metacomputation: Metasystem transitions plus supercompilation. In O. Danvy, R. Glück, and P. Thiemann, editors, *Proceedings of the 1996 Dagstuhl Seminar on Partial Evaluation*, LNCS 1110, pages 482–509, Schloß Dagstuhl, 1996. Springer-Verlag.

[55] P. Wadler. Deforestation: Transforming programs to eliminate intermediate trees. *Theoretical Computer Science*, 73:231–248, 1990. Preliminary version in ESOP'88, LNCS 300.