Constrained Partial Deduction

Michael Leuschel and Danny De Schreye
K.U. Leuven, Department of Computer Science
Celestijnenlaan 200 A, B-3001 Heverlee, Belgium
e-mail: {michael,dannyd}@cs.kuleuven.ac.be

Abstract

Partial deduction based upon the Lloyd and Shepherdson framework generates a specialised program given a set of atoms. Each such atom represents *all* its instances. This can severely limit the specialisation potential of partial deduction. We therefore extend the precision the Lloyd and Shepherdson approach by integrating ideas from constraint logic programming. We formally prove correctness of this new framework of *constrained partial deduction* and illustrate its potential on some examples.

1 Partial Deduction

In contrast to ordinary (full) evaluation, a *partial evaluator* is given 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 input D, produces the same output as the original program P. The goal is to exploit the static input in order to derive more efficient programs. To obtain the specialised program P_S, a partial evaluator performs a mixture of evaluation, i.e. it executes those parts of P which only depend on the static input S, and of code generation for those parts of P which require the dynamic input D. Because part of the computation has already been performed beforehand by the partial evaluator, the hope that we obtain a more efficient program P_S seems justified.

Partial evaluation has received considerable attention in logic programming [9, 18, 29] and functional programming (see e.g. [15] 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 $P \cup \{G\}$. For partial evaluation, 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 [16, 17] (introductions can be found in [18, 9, 4, 21]). Its general idea is to construct a finite set of atoms $\mathbf{A}$ and a finite set of finite, but possibly *incomplete* SLDNF-tree[1] (one for every[2] atom in $\mathbf{A}$) 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 [27], Lloyd and Shepherdson presented a fundamental correctness theorem for partial deduction. The two (additional) basic requirements for correctness of a partial deduction of P wrt $\mathbf{A}$ are the *independence* and *closedness* conditions. The independence condition guarantees that the specialised program does not produce additional answers and the closedness condition guarantees that all calls, which might occur during the execution of the specialised program, are covered by some definition.

2 Motivations

One drawback of (ordinary) partial deduction is that every atom in $\mathbf{A}$ stands for *all* of its instances. As identified in [22] this limits the precision and specialisation that a concrete partial deduction algorithm can attain.

The basic idea of [22], which we summarise in this paper, is to move to a setting of *constrained partial deduction*, which will allow us to enhance precision and specialisation by incorporating constraints. More precisely, constrained partial deduction works on a set $\mathcal{A}$ of *constrained atoms*: couples of the form $c \Box A$ consisting of a constraint c and an atom A.

The richer possibilities conferred by the use of the constraints notably allows one to e.g. provide very precise control mechanisms, "drive negative information" (using the terminology of supercompilation [34, 35]), handle built-ins (like $</2, \backslash== /2$) much more precisely and even make use of type information or argument size relations. We elaborate on two of these motivations below. We will formally present the framework of constrained partial deduction and summarise correctness results in Section 3.

2.1 Preserving Characteristic Trees

A difficult aspect of the control of partial deduction is to decide for which atoms specialised predicate definitions should be produced (i.e. how does one obtain the set $\mathbf{A}$ mentioned above). A refined approach is the one based on *characteristic trees* [10, 8]. Indeed, an abstraction operator like the most specific generalisation[3] (*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, 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.

---

[1] As common in partial deduction, the notion of SLDNF-trees is extended to also allow *incomplete* SLDNF-trees which, in addition to success and failure leaves, may also contain leaves where no literal has been selected for a further derivation step. Leaves of the latter kind will be called *dangling* [28]. Also, a *trivial* SLDNF-tree is one whose root is a dangling leaf.

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

[3] Also known as anti-unification or least general generalisation, see e.g. [19].

Unfortunately, as shown in [22], 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.

Example 2.1 Let P be the program:

(1) $\quad p(X) \leftarrow$

(2) $\quad p(c) \leftarrow$

Take the set of atoms $\mathbf{A} = \{p(a), p(b)\}$. One can see that both $p(a)$ and $p(b)$ specialise in the same way: neither can be resolved with clause (2) and both resolve successfully with clause (1). Using the terminology of [10, 8], $p(a)$ and $p(b)$ have the same characteristic tree. The techniques of [10, 8] would therefore abstract $p(a)$ and $p(b)$ by the more general atom $p(X)$. Unfortunately, $p(X)$ has a different specialisation behaviour: it can be resolved with clause (2) and the pruning — an essential aspect of specialisation — that was possible for the atoms $p(a)$ and $p(b)$ has been lost. More importantly there exists *no* atom, more general than $p(a)$ and $p(b)$, which preserves this pruning.

However, based on the more expressive and powerful framework of constrained partial deduction, one can develop a precise and terminating specialisation algorithm, preserving characteristic trees upon generalisation. The details of this algorithm, which uses constraints expressed in $\mathcal{FT}$, consisting of Clark's equality theory (CET) over the domain of *finite trees*, can be found in [22].

The basic idea of the method is as follows. When taking the *msg* of two (constrained) atoms A and B with the same characteristic tree τ, we do not necessarily obtain an atom C which has the same characteristic tree. Instead of C, the method generates $c \,\square\, C$ as the generalisation of A and B, where the constraint c is designed in such a way as to prune the possible computations of C into the right shape, namely τ. Indeed, all the derivations that were possible for A and B are also possible for C (only definite programs and goals are considered) and c only has to ensure that the additional matches wrt τ are pruned off at some point (cf. Figure 1). In the context of Example 2.1 one would generalise $p(a)$ and $p(b)$ by $X \neq c \,\square\, p(X)$. As we will see later, this constrained atom will not resolve with clause (1) of Example 2.1 and no pruning has been lost due to the abstraction.

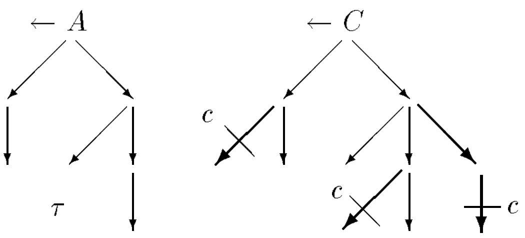

Figure 1: Pruning Constraints

An automatic constrained partial deduction system, based on this idea, has been developed in order to check feasibility as well as practical potential of the approach. The results are promising and can be found in [22]. See also [20, 25, 21] for an alternate approach which preserves characteristic trees upon generalisation.

2.2 Driving of Negative Information

By using constraints over integers or reals, one can handle Prolog built-ins like $<, >, \leq, \geq$ in a much more sophisticated manner than ordinary partial evaluators. Also, one can provide a very refined treatment of the \== Prolog built-in using the $\mathcal{FT}$ structure (this feature has actually been incorporated in the prototype of [22]). The following example illustrates this, where a form of "driving of negative information" (using the terminology of supercompilation [34, 35]) is achieved by constrained partial deduction.

Example 2.2 Take the following adaptation of the *member* program which only succeeds once.

(1) $member(X, [X|T]) \leftarrow$

(2) $member(X, [Y|T]) \leftarrow X \backslash == Y, member(X, T)$

Let us start specialisation with the goal $\leftarrow member(X, [a, Y, a])$. Using a determinate [9] unfolding rule (even with a lookahead) we would unfold this goal once and get the resolvent $\leftarrow X \backslash == a, member(X, [Y, a])$ in the second branch. Ordinary partial deduction would ignore $X \backslash == a$ and unfold $member(X, [Y, a])$, thus producing an extra superfluous resultant with the impossible (given the context) computed answer $\{X/a\}$. In the constrained partial deduction setting, we can incorporate $X \backslash == a$ as a constraint and unfold $\neg(X = a) \Box member(X, [Y, a])$ instead of just $member(X, [Y, a])$ and thereby prune the superfluous resultant.

3 Constrained Partial Deduction

In this section we present the framework of constrained partial deduction and present a correctness result. First, however, to formalise constraints and their effect, we need some basic terminology from *constraint logic programming (CLP)* [14]. From now on we will restrict ourselves to definite programs and goals. We assume basic familiarity with logic programming [1, 26].

3.1 Constraint Logic Programming

First, the predicate symbols are partitioned into two disjoint sets Π_c (the predicate symbols to be used for constraints, notably including "=") and Π_b (the predicate symbols for user-defined predicates). The signature Σ contains all predicate and function symbols with their associated arity. A *constraint* is a first-order formula whose predicate symbols are all contained in Π_c. A formula, atom or literal whose predicate symbols are all contained in Π_b will be called *ordinary*. We will often use the connective " $\Box$ " (and as usual in standard logic programming ",") in the place of "$\wedge$". A *CLP-goal* is denoted by $\leftarrow c \Box B_1, \ldots, B_n$, where c is a constraint and $B_1, \ldots, B_n$ are ordinary atoms. A *CLP-clause* is denoted by $H \leftarrow c \Box B_1, \ldots, B_n$, where c is a constraint and $H, B_1, \ldots, B_n$ are ordinary atoms. Note that, although we do not allow negation within $H, B_1, \ldots, B_n$, negation can still be used within the constraint c. A *CLP-program* is a set of CLP-clauses. Note that CLP-programs will only be required as an intermediary step, the initial and the final specialised programs will be ordinary programs.

The semantics of constraints is given by a Σ-*structure* $\mathcal{D}$, consisting of a domain D and an assignment of functions and relations on D to function symbols in Σ and to predicate symbols in Π_c. Given a constraint c, we will denote by $\mathcal{D} \models c$ the fact that c is true under the interpretation provided by $\mathcal{D}$. Also, c will be called *$\mathcal{D}$-satisfiable* iff $\mathcal{D} \models \tilde{\exists}(c)$, where $\tilde{\exists}(F)$ denotes the existential closure of a formula F. We will also use the standard notation $\tilde{\forall}(F)$ for the universal closure of a first-order formula F and $\tilde{\exists}_{\mathcal{V}}(F)$ (respectively $\tilde{\forall}_{\mathcal{V}}(F)$) for the existential (respectively universal) closure of F except for the variables in the set $\mathcal{V}$.

Applying a substitution on a constraint is defined inductively as follows:

- $p(\bar{t})\theta = p(\bar{t}\theta)$ for $p \in \Pi_c$
- $(F \circ G)\theta = F\theta \circ G\theta$ for $\circ \in \{\wedge, \vee, \leftarrow, \rightarrow, \leftrightarrow\}$.
- $(\neg F)\theta = \neg(F\theta)$
- $(\Pi X.F)\theta = \Pi X'.(F\theta')$ where X' is a new fresh variable not occurring in F and θ, and where $\theta' = \{X/X'\} \cup \{x/t \mid x/t \in \theta \wedge x \neq X\}$ for $\Pi \in \{\forall, \exists\}$.

Applying a substitution on a constraint is used to make explicit the fact that certain variables are determined. For example, $(\forall X.\neg(Y = f(X)))\{Y/g(X)\} = \forall Z.\neg(g(X) = f(Z))$. The following notations for constraints and CLP-goals will also prove useful:

- $holds_{\mathcal{D}}(c) =_{\mathrm{Def}} \mathcal{D} \models \tilde{\forall}(c)$,
- $\theta\ sat_{\mathcal{D}}\ c =_{\mathrm{Def}} holds_{\mathcal{D}}(c\theta)$.
- $vars(c) =_{\mathrm{Def}}$ the free variables in c.
- $vars(\leftarrow c \,\Box\, Q) =_{\mathrm{Def}} vars(c) \cup vars(Q)$.

Note that for CLP-goals $\leftarrow c \,\Box\, Q$ we will in fact require that $vars(c) \subseteq vars(Q)$[4] (meaning that actually $vars(\leftarrow c \,\Box\, Q) = vars(Q)$). This will be ensured by applying the existential closure $\tilde{\exists}_{vars(Q)}(.)$ during derivation steps below (this existential closure makes no difference wrt $\mathcal{D}$-satisfiability, but it makes a difference wrt $holds_{\mathcal{D}}$).

We now define a counterpart to SLD-derivations for CLP-goals. In our context of partial deduction, the initial and final programs are just ordinary logic programs. In order for our constraint manipulations to be correct wrt the initial *ordinary* logic program, we have to ensure that equality is not handled in an unsound manner. For instance, something like $a = b$ should not succeed. In other words, if there is no SLD-refutation for $P \cup \{\leftarrow Q\}$ then there should be no CLP-refutation for any $P \cup \{\leftarrow c \,\Box\, Q\}$ either. To ensure this property we use the following definition of a derivation, adapted from [7], in which substitutions are made explicit. This will also enable us to construct resultants in a straightforward manner.

Definition 3.1 *Let $CG = \leftarrow c \,\Box\, L_1, \ldots, L_k$ a CLP-goal and $C = A \leftarrow B_1, \ldots, B_n$ a program clause*[5] *such that $k \geq 1$ and $n \geq 0$. Then CG'* is derived from *CG* and *C* in *$\mathcal{D}$ using θ iff the following conditions hold:*

- *L_m is an atom, called the* selected *atom (at position m), in CG.*
- *θ is a relevant and idempotent mgu of L_m and A.*
- *CG' is the goal $\leftarrow c' \,\Box\, Q$, where $Q = (L_1, \ldots, L_{m-1}, B_1, \ldots, B_n, L_{m+1}, \ldots, L_k)\theta$ and $c' = \tilde{\exists}_{vars(Q)}(c\theta)$.*
- *c' is $\mathcal{D}$-satisfiable.*

CG' is called a resolvent *of CG and C in $\mathcal{D}$.*

Definition 3.2 (complete CLP$_=(\mathcal{D})$-derivation) *Let P be a definite program and CG_0 a CLP-goal. A* complete CLP$_=(\mathcal{D})$-derivation *of $P \cup \{CG_0\}$ is a tuple $(\mathcal{G}, \mathcal{C}, \mathcal{S})$ consisting of a (finite or infinite) sequence of CLP-goals $\mathcal{G} = \langle CG_0, CG_1, \ldots\rangle$, a sequence $\mathcal{C} = \langle C_1, C_2, \ldots\rangle$ of variants of program clauses of P and a sequence $\mathcal{S} = \langle \theta_1, \theta_2, \ldots\rangle$ of mgu's such that:*

[4] As the conjunction Q contains no quantifiers, $vars(Q)$ are the free variables of Q.

[5] It is possible to extend the above definition (and the following ones) for CLP-clauses, but such an extension is not needed for the formalisation of the framework.

- *for* $i > 0$, $vars(C_i) \cap vars(CG_0) = \emptyset$;
- *for* $i > j$, $vars(C_i) \cap vars(C_j) = \emptyset$;
- *for* $i \geq 0$, CG_{i+1} *is derived from* CG_i *and* C_{i+1} *in* $\mathcal{D}$ *using* θ_{i+1} *and*
- *the sequences* $\mathcal{G}, \mathcal{C}, \mathcal{S}$ *are maximal (given the choice of the selected atoms).*

A *CLP*$_=(\mathcal{D})$*-refutation* is just a complete CLP$_=(\mathcal{D})$-derivation whose last goal contains no atoms, i.e. it can be written as $\leftarrow c \,\Box\, \epsilon$ where ϵ denotes the empty sequence of atoms. A *finitely failed CLP*$_=(\mathcal{D})$*-derivation* is a finite, complete CLP$_=(\mathcal{D})$-derivation whose last goal is *not* of the form $\leftarrow c \,\Box\, \epsilon$. There are thus 3 forms of complete CLP$_=(\mathcal{D})$-derivations: refutations, finitely failed ones and infinite derivations.

In the context of partial deduction we also allow incomplete derivations. A CLP$_=(\mathcal{D})$-derivation is defined like a complete CLP$_=(\mathcal{D})$-derivation but may, in addition to leading to success or failure, also lead to a last goal where no atom has been selected for a further derivation step. Derivations of the latter kind will be called *incomplete*.

The concept of CLP$_=(\mathcal{D})$-trees can be defined just like the concept of SLD-trees: its branches are just CLP$_=(\mathcal{D})$-derivations instead of SLD-derivations.

In order to construct resultants we also need the following, where $\theta|_{\mathcal{V}}$ denotes the restriction of the substitution θ to the set of variables $\mathcal{V}$:

Definition 3.3 *The* computed answer *of a finite, non-failed CLP*$_=(\mathcal{D})$*-derivation* δ *for* $P \cup \{\leftarrow c \,\Box\, G\}$ *with the sequence* $\theta_1, \ldots, \theta_n$ *of mgu's, is the substitution* $cas(\delta) = (\theta_1 \ldots \theta_n)|_{vars(G)}$. *Also, the last goal of* δ *will be called the* resolvent *of* δ.

3.2 A Framework for Constrained Partial Deduction

We will now present a generic partial deduction scheme which, instead of working on sets of ordinary atoms, will work on sets of *constrained atoms*.

Definition 3.4 *A* constrained atom *is formula of the form* $c \,\Box\, A$ *where* c *is a constraint and* A *an ordinary atom such that the free variables of* c *are contained in the variables of* A.

Definition 3.5 ($valid_{\mathcal{D}}$) *Let* $c \,\Box\, A$ *be a constrained atom. The set of* valid $\mathcal{D}$-instances *of* $c \,\Box\, A$ *is defined as:* $valid_{\mathcal{D}}(c \,\Box\, A) = \{A\theta \mid \theta \ sat_{\mathcal{D}} \ c\}$.

By definition of $sat_{\mathcal{D}}$, the set of valid $\mathcal{D}$-instances is *downwards-closed* (or closed under substitution, i.e. if $A' \in valid_{\mathcal{D}}(c \,\Box\, A)$ then so is any instance of A'). The constraint within a constrained atom thus specifies a property that holds for all valid instances, which in our case correspond to the possible runtime instances.

We also need an instance notion on constrained atoms.

Definition 3.6 **($\mathcal{D}$-instance)** *Let* $c \,\Box\, A$, $c' \,\Box\, A'$ *be constrained atoms. Then* $c' \,\Box\, A'$ *is a* $\mathcal{D}$-instance *of* $c \,\Box\, A$, *denoted by* $c' \,\Box\, A' \preceq_{\mathcal{D}} c \,\Box\, A$, *iff* $A' = A\gamma$ *and* $valid_{\mathcal{D}}(c' \,\Box\, A') \subseteq valid_{\mathcal{D}}(c \,\Box\, A)$.

For example, independently of $\mathcal{D}$, $\neg(X = c) \,\Box\, p(X)$ is a $\mathcal{D}$-instance of $true \,\Box\, p(X)$ because every substitution satisfies $true$. In turn, if $\mathcal{D}$ is e.g. Clark's equality theory (CET, see e.g. [3, 26]) then $true \,\Box\, p(b)$ is a $\mathcal{D}$-instance of $\neg(X = c) \,\Box\, p(X)$ because $\{X/b\} \ sat_{\mathcal{D}} \ \neg(X = c)$ (i.e. $CET \models \tilde{\forall}(\neg(b = c))$).

Definition 3.7 (partial deduction of $c \Box A$) *Let P be a program and $c \Box A$ a constrained atom. Let τ be a finite, non-trivial and possibly incomplete CLP$_=(\mathcal{D})$-tree for $P \cup \{\leftarrow c \Box A\}$ and let $\leftarrow c_1 \Box G_1, \ldots, \leftarrow c_n \Box G_n$ be the CLP-goals in the leaves of this tree. Let $\theta_1, \ldots, \theta_n$ be the computed answers of the CLP$_=(\mathcal{D})$-derivations from $\leftarrow c \Box A$ to $\leftarrow c_1 \Box G_1, \ldots, \leftarrow c_n \Box G_n$ respectively. Then the set of CLP-resultants $\{A\theta_1 \leftarrow c_1 \Box G_1, \ldots, A\theta_n \leftarrow c_n \Box G_n\}$ is called the* partial deduction of $c \Box A$ in P (using $\mathcal{D}$).

Example 3.8 Take the program P from Example 2.1.

(1) $p(X) \leftarrow$
(2) $p(c) \leftarrow$

When using the constraint structure $\mathcal{D}$ = CET (or any other structure in which $\neg(c = c)$ is unsatisfiable), a partial deduction of $\neg(X = c) \Box p(X)$ in P (using $\mathcal{D}$[6]) is:

(1') $p(X) \leftarrow \neg(X = c) \Box \epsilon$

We now generate partial deductions not for sets of atoms, but for sets of *constrained* atoms. As such, the same atom A might occur in several constrained atoms but with different associated constraints. This means that *renaming* (i.e. mapping [constrained] atoms with common instances to new predicate symbols) as a way to ensure independence (see e.g. [22, 21]) imposes itself even more than in the standard partial deduction setting. In addition to renaming, we will also allow argument filtering (i.e. filtering out constants and functors and only keep the variables as arguments.), leading to the following definition.

First, given a CLP-clause $C = H \leftarrow c \Box B_1, \ldots, B_n$, each constrained atom of the form $\tilde{\exists}_{vars(B_i)}(c) \Box B_i$ will be called a *constrained body atom of C*. This notion extends to CLP-programs by taking the union of the constrained body atoms of the clauses.

Definition 3.9 (atomic renaming, renaming function) *An* atomic renaming α *for a set $\mathcal{A}$ of constrained atoms maps each constrained atom in $\mathcal{A}$ to an atom such that*

- *for each $c \Box A \in \mathcal{A}$: $vars(\alpha(c \Box A)) = vars(A)$*
- *for $CA, CA' \in \mathcal{A}$ such that $CA \neq CA'$: the predicate symbols of $\alpha(CA)$ and $\alpha(CA')$ are distinct (but may occur in $\mathcal{A}$).*

Let P be a program. A renaming function ρ_α *for $\mathcal{A}$ based on α is a mapping from constrained atoms to atoms such that:*

$$\rho_\alpha(c \Box A) = \alpha(c' \Box A')\theta \text{ for some } c' \Box A' \in \mathcal{A} \text{ with } A = A'\theta \wedge \;\; c \Box A \preceq_{\mathcal{D}} c' \Box A'.$$

We leave $\rho_\alpha(A)$ undefined if $c \Box A$ is not a $\mathcal{D}$-instance of an element in $\mathcal{A}$.
A renaming function ρ_α can also be applied to constrained goals $c \Box B_1, \ldots, B_n$, by applying it individually to each constrained body atom $c_i \Box B_i$. Finally, we can apply a renaming function also to ordinary goals by defining $\rho_\alpha(G) = \rho_\alpha(true \Box G)$.

Note that if the set of $\mathcal{D}$-instances of two or more elements in $\mathcal{A}$ overlap then ρ_α must make a choice for the atoms in the intersection of the concretisations and several renaming functions based on the same α exist.

Definition 3.10 (partial deduction wrt $\mathcal{A}$) *Let P be a program, $\mathcal{A} = \{c_1 \Box A_1, \ldots, c_n \Box A_n\}$ be a finite set of constrained atoms and let ρ_α be a renaming for $\mathcal{A}$ based on the atomic renaming α. For each $i \in \{1, \ldots, n\}$, let R_i be the partial deduction of $c_i \Box A_i$ in*

[6] We will often take the liberty to not always explicitly mention the constraint domain $\mathcal{D}$ which was used to construct partial deductions and assume that $\mathcal{D}$ is fixed and known.

P and let $\hat{P} = \{R_i \mid i \in \{1, \ldots, n\}\}$. Then the program $\{\alpha(c_i \Box A_i)\theta \leftarrow \rho_\alpha(c \Box Bdy) \mid A_i\theta \leftarrow c \Box Bdy \in R_i \wedge 1 \leq i \leq n \wedge \rho_\alpha(c \Box Bdy)$ is defined $\}$ is called the partial deduction of P wrt $\mathcal{A}$, $\hat{P}$ and ρ_α.

Example 3.11 Let P be the program of Example 3.8. Also let us use the same constraint structure $\mathcal{D}$ as in Example 3.8. In the context of P, we can abstract the constrained atoms $true \Box p(a)$ and $true \Box p(b)$ by the more general constrained atom $\neg(X = c) \Box p(X)$, having the same specialisation behaviour (something which is impossible to attain in ordinary partial deduction, see Example 2.1). As illustrated in Example 3.8, the additional match with clause (2) is pruned for $\neg(X = c) \Box p(X)$, because $\neg(X = c)\{X/c\}$ is unsatisfiable in $\mathcal{D}$. The partial deduction of $\neg(X = c) \Box p(X)$ based on $\alpha(\neg(X = c) \ \Box p(X)) = p'(X)$ is thus

(1) $\quad p'(X) \leftarrow$

Note that $\rho_\alpha(\leftarrow \neg(X = c) \Box \epsilon) = \epsilon$, i.e. the empty goal. The renaming of the run-time goal $\leftarrow p(a), p(b)$ is $\leftarrow p'(a), p'(b)$.

Note that the partial deduction wrt $\mathcal{A}$ is an ordinary logic program *without constraints*. The coveredness criterion presented in the next subsection, will ensure that the constraint manipulations have already been incorporated (by pruning certain resultants) and no additional constraint processing at run-time is needed.

3.3 Correctness of Constrained Partial Deduction

Let us first rephrase the coveredness condition of standard partial deduction in the context of constrained atoms. This definition will also ensure that the renamings, applied for instance in Definition 3.10, are always defined.

Definition 3.12 *Let $\hat{P}$ be a CLP-program and $\mathcal{A}$ a set of constrained atoms. Then $\hat{P}$ is called* $\mathcal{A}, \mathcal{D}$-covered *iff each of its constrained body atoms is a $\mathcal{D}$-instance of a constrained atom in $\mathcal{A}$.*

We can extend the above notion also to ordinary programs and goals by inserting the constraint *true* (e.g. $H \leftarrow Bdy$ is $\mathcal{A}, \mathcal{D}$-covered iff $H \leftarrow true \Box Bdy$ is).

The main correctness result for constrained partial deduction is as follows.

Theorem 3.13 *Let P be a definite program, G a definite goal, $\mathcal{A}$ a finite set of constrained atoms, ρ_α a renaming function for $\mathcal{A}$ based on α and P' the partial deduction of P wrt $\mathcal{A}$, $\hat{P}$ and ρ_α. If $\hat{P} \cup \{G\}$ is $\mathcal{A}, \mathcal{D}$-covered then the following hold:*

1. *$P' \cup \{\rho_\alpha(G)\}$ has an SLD-refutation with computed answer θ iff $P \cup \{G\}$ does.*
2. *$P' \cup \{\rho_\alpha(G)\}$ has a finitely failed SLD-tree iff $P \cup \{G\}$ does.*

The proof can be found in [22].

4 Discussion and Conclusion

A related work is [5], which uses abstract substitutions to prune resultants while unfolding. These abstract substitutions play a role very similar to the constraints in the current paper.

However, no formal correctness or termination result is given in [5] (and the issue of preserving characteristic trees is not addressed). Indeed, as abstract substitutions of [5] are not necessarily downwards-closed, this seems to be a much harder task and a normal coveredness condition will not suffice to ensure correctness (for instance the atoms in the bodies of clauses might be further instantiated at run-time and thus, in the absence of downwards-closedness, no longer covered). Our paper actually provides a framework within which correctness of [5] could be established for abstract substitutions which are downwards-closed. Another, more technical difference is that neither the method of [6, 11] nor the method of [5] preserve the finite failure semantics (i.e. infinite failure might be replaced by finite failure), while our approach, just like ordinary partial deduction, does.

Another method that might look like a viable alternative to our approach is the one of [2], situated within the context of unfold/fold transformations. In particular, [2] contains many transformation rules and allows first-order logic formulas to be used to constrain the specialisation. It is thus a very powerful framework. But also, because of that power, controlling it in an automatic way, as well as ensuring actual efficiency gains, is much more difficult. A prototype for [2] exists, but the control heuristics as well as the correctness proofs are still left to the user.

The program treated e.g. in Example 2.2 is actually almost a CLP-program, and we could go one step further and also specialise CLP-programs. As the concrete algorithm of [22], briefly described in Section 2.1, is based on the structure $\mathcal{FT}$ we conjecture that an adaptation of our technique might yield a refined specialisation technique for CLP($\mathcal{FT}$) [30, 31, 32]. Also, it is actually not very difficult to adapt the framework of Section 3.2 to work on CLP-programs instead of ordinary logic programs — we just have to require that equality is handled in the same manner as in logic programming. However, establishing the correctness will become much more difficult because one cannot reuse the correctness results of standard partial deduction. In that context, we would like to mention [33], which extends constructive negation for CLP-programs, as well as recent work on the transformation of CLP-programs [7]. Note, however, that [7] is situated within the unfold/fold transformation paradigm and also that no concrete algorithms are presented.

Finally, let us mention a recent extension of partial deduction, called conjunctive partial deduction [24, 13, 21]. Conjunctive partial deduction handles conjunctions of atoms instead of just atoms. We believe that the framework of constrained partial deduction can be extended to handle conjunctions as well and that both frameworks will benefit from the increased capabilities.[7]

In conclusion, to overcome several inadequacies of classical partial deduction we have developed the framework of *constrained partial deduction*, based on introducing constraints into the partial deduction process. We have provided formal correctness results for this framework and have shown that it offers increased potential.

Acknowledgements

Michael Leuschel is supported by the Belgian GOA "Non-Standard Applications of Abstract Interpretation". Danny De Schreye is senior research associate of the Belgian National Fund for Scientific Research. We would like to thank Bern Martens for proof-reading (several versions) of this paper,

[7] First investigations indicate for instance that, together with the technique of [23], sophisticated inductive theorems can be proven (the relation between program specialisation and theorem proving has already been raised several times in the literature [35, 12, 36]).

for his subtle comments and for the stimulating discussions. We would also like to thank John Gallagher and Maurice Bruynooghe for their helpful remarks. We appreciated interesting discussions with Włodek Drabent, Jesper Jørgensen and André De Waal. We are also grateful to Marc Denecker for providing us with relevant insights into semantical issues and equality theory. Finally we thank anonymous referees of New Generation Computing and WLP'97 for their comments.

From: Michael Leuschel ¡Michael.Leuschel@cs.kuleuven.ac.be¿ Date: Wed, 20 Aug 1997 15:30:48 +0200 (MET DST) To: seipel@informatik.uni-wuerzburg.de Subject: BBL Datei fuer Constrained Partial Deduction

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] A. Bossi, N. Cocco, and S. Dulli. A method for specialising logic programs. *ACM Transactions on Programming Languages and Systems*, 12(2):253–302, 1990.

[3] K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

[4] 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.

[5] D. A. de Waal and J. Gallagher. Specialisation of a unification algorithm. In T. Clement and K.-K. Lau, editors, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'91*, pages 205–220, Manchester, UK, 1991.

[6] D. A. de Waal and J. Gallagher. The applicability of logic program analysis and transformation to theorem proving. In A. Bundy, editor, *Automated Deduction—CADE-12*, pages 207–221. Springer-Verlag, 1994.

[7] S. Etalle and M. Gabbrielli. A transformation system for modular CLP programs. In L. Sterling, editor, *Proceedings of the 12th International Conference on Logic Programming*, pages 681–695. The MIT Press, 1995.

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

[9] 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.

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

[11] J. Gallagher and D. A. de Waal. Deletion of redundant unary type predicates from logic programs. In K.-K. Lau and T. Clement, editors, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'92*, pages 151–167, Manchester, UK, 1992.

[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. Extended version as Technical Report CW 226, K.U. Leuven. Accessible via `http://www.cs.kuleuven.ac.be/~lpai`.

[14] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *The Journal of Logic Programming*, 19 & 20:503–581, 1994.

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

[16] J. Komorowksi. *A Specification of an Abstract Prolog Machine and its Application to Partial Evaluation*. PhD thesis, Linköping University, Sweden, 1981. Linköping Studies in Science and Technology Dissertations 69.

[17] 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.

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

[19] 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.

[20] 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.

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

[22] M. Leuschel and D. De Schreye. Constrained partial deduction and the preservation of characteristic trees. Technical Report CW 250, Departement Computerwetenschappen, K.U. Leuven, Belgium, June 1997. Accepted for Publication in New Generation Computing. Accessible via `http://www.cs.kuleuven.ac.be/~lpai`.

[23] 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. Accessible via `http://www.cs.kuleuven.ac.be/~lpai`.

[24] 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. Accessible via `http://www.cs.kuleuven.ac.be/~lpai`.

[25] 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. Accessible via `http://www.cs.kuleuven.ac.be/~lpai`.

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

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

[28] B. Martens and D. De Schreye. Automatic finite unfolding using well-founded measures. *The Journal of Logic Programming*, 28(2):89–146, August 1996. Abridged and revised version of Technical Report CW180, Departement Computerwetenschappen, K.U.Leuven, October 1993, accessible via `http://www.cs.kuleuven.ac.be/~lpai`.

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

[30] D. A. Smith. Constraint operations for CLP($\mathcal{FT}$). In K. Furukawa, editor, *Logic Programming: Proceedings of the Eighth International Conference*, pages 760–774. MIT Press, 1991.

[31] D. A. Smith. Partial evaluation of pattern matching in constraint logic programming languages. In N. D. Jones and P. Hudak, editors, *ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 62–71. ACM Press Sigplan Notices 26(9), 1991.

[32] D. A. Smith and T. Hickey. Partial evaluation of a CLP language. In S. Debray and M. Hermenegildo, editors, *Proceedings of the North American Conference on Logic Programming*, pages 119–138. MIT Press, 1990.

[33] P. J. Stuckey. Negation and constraint logic programming. *Information and Computation*, 118(1):12–33, April 1995.

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

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

[36] 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.