Logic Program Specialisation

Michael Leuschel

Department of Electronics and Computer Science
University of Southampton
Highfield, Southampton, SO17 1BJ, UK
mal@ecs.soton.ac.uk
www: http://www.ecs.soton.ac.uk/~mal

1 Introduction

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. *Logic programming* and *functional programming* are two prominent members of this class of programming languages. While functional programming is based on the λ-calculus, logic programming has its roots in first-order logic and automated theorem proving. Both approaches share the view that a program is a *theory* and execution consists in performing *deduction* from that theory.

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 can be used to speed up existing programs for certain application domains, sometimes achieving speedups of several orders of magnitude. It, however, also 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.

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. First, because there *exists* a *clear* and *simple* semantical foundation, techniques for program specialisation *can* be proven correct in a formal way. Furthermore, program specialisation does not have to preserve every execution aspect of the source program, as long as the declarative semantics is respected. This permits much more powerful optimisations, impossible to obtain when the specialiser has to preserve every operational aspect of the source program.

This course is situated within that context and is structured as follows. Section 2 starts out from the roots of logic programming in first-order logic and automated theorem proving and presents the syntax, semantics and proof theory of logic programs. In Section 3 the general idea of program specialisation, based on Kleene's S-M-N theorem, is introduced. A particular technique for specialising logic programs, called *partial deduction*, is then developed and illustrated. The theoretical underpinnings of this approach, based on the correctness results by Lloyd and Shepherdson [69], are exhibited. We also elaborate on the control issues of partial deduction and define the *control of polyvariance problem.*

The advanced part of this course [60] (in this volume), builds upon these foundations and presents more refined techniques for controlling partial deduction, as well as several ways of extending its power, all situated within the larger objective of turning declarative languages and program specialisation into valuable tools for constructing reliable, maintainable *and* efficient programs.

2 Logic and Logic Programming

In this section we summarise some essential background in first-order logic and logic programming. Not every detail is required for the proper comprehension of this course and this section is mainly meant to be a kind of "reference manual" of logic programming.

The exposition is mainly inspired by [5] and [68] and in general adheres to the same terminology. The reader is referred to these works for a more detailed presentation, comprising motivations, examples and proofs. Some other good introductions to logic programming can also be found in [78] and [33, 6], while a good introduction to first-order logic and automated theorem proving can be found in [36].

2.1 First-order logic and syntax of logic programs

We start with a brief presentation of first-order logic.

Definition 1. (alphabet) *An* alphabet *consists of the following classes of symbols:* variables*;* function symbols*;* predicate symbols*;* connectives*, which are $\neg$ negation, $\wedge$ conjunction, $\vee$ disjunction, $\leftarrow$ implication, and $\leftrightarrow$ equivalence;* quantifiers*, which are the existential quantifier $\exists$ and the universal quantifier $\forall$;* punctuation symbols*, which are "(", ")" and ",". Function and predicate symbols have an associated* arity*, a natural number indicating how many arguments they take in the definitions following below.*
Constants *are function symbols of arity 0, while* propositions *are predicate symbols of arity 0.*

In the remainder of this course we suppose the set of variables is countably infinite. In addition, alphabets with a finite set of function and predicate symbols will simply be called *finite*. An *infinite* alphabet is one in which the number of function and/or predicate symbols is not finite but countably infinite.

We will try to adhere as much as possible to the following syntactical conventions throughout the course:
- Variables will be denoted by upper-case letters like X, Y, Z, usually taken from the later part of the (Latin) alphabet.
- Constants will be denoted by lower-case letters like a, b, c, usually taken from the beginning of the (Latin) alphabet.
- The other function symbols will be denoted by lower-case letters like f, g, h.
- Predicate symbols will be denoted by lower-case letters like p, q, r.

Definition 2. (terms, atoms) *The set of* terms *(over some given alphabet) is inductively defined as follows:*

- *a variable is a term*
- *a constant is a term and*
- *a function symbol f of arity $n > 0$ applied to a sequence $t_1, \ldots, t_n$ of n terms, denoted by $f(t_1, \ldots, t_n)$, is also a term.*

The set of atoms *(over some given alphabet) is defined in the following way:*

- *a proposition is an atom and*
- *a predicate symbol p of arity $n > 0$ applied to a sequence $t_1, \ldots, t_n$ of n terms, denoted by $p(t_1, \ldots, t_n)$, is an atom.*

We will also allow the notations $f(t_1, \ldots, t_n)$ and $p(t_1, \ldots, t_n)$ in case $n = 0$. $f(t_1, \ldots, t_n)$ then simply represents the term f and $p(t_1, \ldots, t_n)$ represents the atom p. For terms representing lists we will use the usual Prolog [28, 95, 23] notation: e.g., $[\,]$ denotes the empty list, $[H|T]$ denotes a non-empty list with first element H and tail T and $[a, b]$ denotes a two-element list made up using a and b.

Definition 3. (formula) *A* (well-formed) formula *(over some given alphabet) is inductively defined as follows:*

- *An atom is a formula.*
- *If F and G are formulas then so are $(\neg F)$, $(F \vee G)$, $(F \wedge G)$, $(F \leftarrow G)$, $(F \leftrightarrow G)$.*
- *If X is a variable and F is a formula then $(\forall X F)$ and $(\exists X F)$ are also formulas.*

To avoid formulas cluttered with the punctuation symbols we give the connectives and quantifiers the following precedence, from highest to lowest:

1. $\neg, \forall, \exists$, 2. $\vee$, 3. $\wedge$, 4. $\leftarrow, \leftrightarrow$.

For instance, we will write $\forall X(p(X) \leftarrow \neg q(X) \wedge r(X))$ instead of the less readable $(\forall X(p(X) \leftarrow ((\neg q(X)) \wedge r(X))))$.

The set of all formulas constructed using a given alphabet A is called the *first-order language* given by A.

First-order logic assigns meanings to formulas in the form of *interpretations* over some domain D:

- Each function symbol of arity n is assigned an n-ary function $D^n \mapsto D$. This part, along with the choice of the domain D, is referred to as a *pre-interpretation*.
- Each predicate symbol of arity n is assigned an n-ary relation, i.e., a subset of D^n (or equivalently an n-ary function $D^n \mapsto \{\mathbf{true}, \mathbf{false}\}$).
- Each formula is given a truth value, **true** or **false**, depending on the truth values of the sub-formulas. (For more details see e.g., [36] or [68]).

A *model* of a formula is simply an interpretation in which the formula has the value **true** assigned to it. Similarly, a model of a set S of formulas is an interpretation which is a model for all $F \in S$.

For example, let I be an interpretation whose domain D is the set of natural numbers $I\!N$ and which maps the constant a to 1, the constant b to 2 and the unary predicate p to the unary relation $\{(1)\}$. Then the truth value of $p(a)$ under I is **true** and the truth value of $p(b)$ under I is **false**. So I is a model of $p(a)$ but not of $p(b)$. I is also a model of $\exists X p(X)$ but not of $\forall X p(X)$.

We say that two formulas are *logically equivalent* iff they have the same models. A formula F is said to be a *logical consequence* of a set of formulas S, denoted by $S \models F$, iff F is assigned the truth value **true** in all models of S. A set of formulas S is said to be *inconsistent* iff it has no model. It can be easily shown that $S \models F$ holds iff $S \cup \{\neg F\}$ is inconsistent. This observation lies at the basis of what is called a proof by *refutation*: to show that F is a logical consequence of S we show that $S \cup \{\neg F\}$ leads to inconsistency.

From now on we will also use **true** (resp. **false**) to denote some arbitrary formula which is assigned the truth value **true** (resp. **false**) in every interpretation. If there exists a proposition p in the underlying alphabet then **true** could, e.g., stand for $p \vee \neg p$ and **false** could stand for $p \wedge \neg p$.[1] We also introduce the following shorthands for formulas:

- if F is a formula, then $(F \leftarrow)$ denotes the formula $(F \leftarrow \mathbf{true})$ and $(\leftarrow F)$ denotes the formula $(\mathbf{false} \leftarrow F)$.
- $(\leftarrow)$ denotes the formula $(\mathbf{false} \leftarrow \mathbf{true})$.

In the following we define some other frequently occurring kinds of formulas.

Definition 4. (literal) *If A is an atom then the formulas A and $\neg A$ are called* literals. *Furthermore, A is called a* positive *literal and $\neg A$ a* negative *literal.*

[1] In some texts on logic (e.g., [36]) **true** and **false** are simply added to the alphabet and treated in a special manner by interpretations.

Definition 5. (conjunction, disjunction) *Let $A_1, \ldots, A_n$ be literals, where $n > 0$. Then $A_1 \wedge \ldots \wedge A_n$ is a* conjunction *and $A_1 \vee \ldots \vee A_n$ is a* disjunction.

Usually we will assume $\wedge$ (respectively $\vee$) to be associative, in the sense that we do not distinguish between the logically equivalent, but syntactically different, formulas $p \wedge (q \wedge r)$ and $(p \wedge q) \wedge r$.

Definition 6. (scope) *Given a formula $(\forall X F)$ (resp. $(\exists X F)$) the* scope *of $\forall X$ (resp. $\exists X$) is F. A* bound occurrence *of a variable X inside a formula F is any occurrence immediately following a quantifier or an occurrence within the scope of a quantifier $\forall X$ or $\exists X$. Any other occurrence of X inside F is said to be* free.

Definition 7. (closure) *Given a formula F, the* universal closure *of F, denoted by $\forall(F)$, is a formula of the form $(\forall X_1 \ldots (\forall X_m F) \ldots)$ where $X_1, \ldots, X_m$ are all the variables having a free occurrence inside F (in some arbitrary order). Similarly the* existential closure *of F, denoted by $\exists(F)$, is the formula $(\exists X_1 \ldots (\exists X_m F) \ldots)$.*

The following class of formulas plays a central role in logic programming.

Definition 8. (clause) *A* clause *is a formula of the form $\forall(H_1 \vee \ldots \vee H_m \leftarrow B_1 \wedge \ldots \wedge B_n)$, where $m \geq 0, n \geq 0$ and $H_1, \ldots, H_m$, $B_1, \ldots, B_n$ are all literals. $H_1 \vee \ldots \vee H_m$ is called the* head *of the clause and $B_1 \wedge \ldots \wedge B_n$ is called the* body. *A* (normal) program clause *is a clause where $m = 1$ and H_1 is an atom. A* definite program clause *is a normal program clause in which $B_1, \ldots, B_n$ are atoms. A* fact *is a program clause with $n = 0$. A* query *or* goal *is a clause with $m = 0$ and $n > 0$. A* definite goal *is a goal in which $B_1, \ldots, B_n$ are atoms. The* empty clause *is a clause with $n = m = 0$. As we have seen earlier, this corresponds to the formula* **false** $\leftarrow$ **true***, i.e., a contradiction. We also use $\Box$ to denote the empty clause.*

In logic programming notation one usually omits the universal quantifiers encapsulating the clause and one also often uses the comma (',') instead of the conjunction in the body, e.g., one writes $p(s(X)) \leftarrow q(X), p(X)$ instead of $\forall X(p(f(X)) \leftarrow (q(X) \wedge p(X)))$. We will adhere to this convention.

Definition 9. (program) *A* (normal) program *is a set of program clauses. A* definite program *is a set of definite program clauses.*

In order to express a given program P in a first-order language L given by some alphabet A, the alphabet A must of course contain the function and predicate symbols occurring within P. The alphabet might however contain additional function and predicate symbols which do not occur inside the program. We therefore denote the underlying first-order language of a given program P by $\mathcal{L}_P$ and the underlying alphabet by $\mathcal{A}_P$. For technical reasons related to definitions below, we suppose that there is at least one constant symbol in $\mathcal{A}_P$.

2.2 Semantics of logic programs

Given that a program P is just a set of formulas, which happen to be clauses, the logical meaning of P might simply be seen as all the formulas F for which $P \models F$. For normal programs this approach will turn out to be insufficient, but for definite programs it provides a good starting point.

Definite programs To determine whether a formula F is a logical consequence of another formula G, we have to examine whether F is true in *all* models of G. One big advantage of clauses is that it is sufficient to look just at certain canonical models, called the Herbrand models.

In the following we will define these canonical models. Any term, atom, literal, clause will be called *ground* iff it contains no variables.

Definition 10. *Let P be a program written in the underlying first-order language $\mathcal{L}_P$ given by the alphabet $\mathcal{A}_P$. Then the* Herbrand universe $\mathcal{U}_P$ *is the set of all ground terms over $\mathcal{A}_P$.*[2] *The* Herbrand base $\mathcal{B}_P$ *is the set of all ground atoms in $\mathcal{L}_P$.*

A *Herbrand interpretation* is simply an interpretation whose domain is the Herbrand universe $\mathcal{U}_P$ and which maps every term to itself. A *Herbrand model* of a set of formulas S is an Herbrand interpretation which is a model of S.

The interest of Herbrand models for logic programs derives from the following proposition (the proposition does *not* hold for arbitrary formulas).

Proposition 1. *A set of clauses has a model iff it has a Herbrand model.*

This means that a formula F which is true in all Herbrand models of a set of clauses C is a logical consequence of C. Indeed if F is true in all Herbrand models then $\neg F$ is false in all Herbrand models and therefore, by Proposition 1, $C \cup \{\neg F\}$ is inconsistent and $C \models F$.

Note that a Herbrand interpretation or model can be identified with a subset H of the Herbrand base $\mathcal{B}_P$ (i.e., $H \in 2^{\mathcal{B}_P}$): the interpretation of $p(d_1, \ldots, d_n)$ is **true** iff $p(d_1, \ldots, d_n) \in H$ and the interpretation of $p(d_1, \ldots, d_n)$ is **false** iff $p(d_1, \ldots, d_n) \notin H$. This means that we can use the standard set order on Herbrand models and define minimal Herbrand models as follows.

Definition 11. *A Herbrand model $H \subseteq \mathcal{B}_P$ for a given program P is a* minimal Herbrand model *iff there exists no $H' \subset H$ which is also a Herbrand model of P.*

For definite programs there exists a *unique* minimal Herbrand model, called the *least Herbrand model*, denoted by $\mathcal{H}_P$. Indeed it can be easily shown that the intersection of two Herbrand models for a definite program P is still a Herbrand model of P. Furthermore, the entire Herbrand base $\mathcal{B}_P$ is always a model for a definite program and one can thus obtain the least Herbrand model by taking the intersection of all Herbrand models.

The least Herbrand model $\mathcal{H}_P$ can be seen as capturing the *intended meaning* of a given definite program P as it is sufficient to infer all the logical consequences of P. Indeed, a formula which is true in the least Herbrand model $\mathcal{H}_P$ is true in all Herbrand models and is therefore a logical consequence of the program.

Example 1. Take for instance the following program P:

$int(0) \leftarrow$
$int(s(X)) \leftarrow int(X)$

Then the least Herbrand model of P is $\mathcal{H}_P = \{int(0), int(s(0)), \ldots\}$ and indeed $P \models int(0)$, $P \models int(s(0))$, ... But also note that for definite programs the entire Herbrand base $\mathcal{B}_P$ is also a model. Given a suitable alphabet $\mathcal{A}_P$, we might have $\mathcal{B}_P = \{int(a), int(0), int(s(a)), int(s(0)), \ldots\}$. This means that the atom $int(a)$ is consistent with the program P (i.e., $P \not\models \neg int(a)$), but is not implied either (i.e., $P \not\models int(a)$).

[2] It is here that the requirement that $\mathcal{A}_P$ contains at least one constant symbol comes into play. It ensures that the Herbrand universe is never empty.

It is here that logic programming goes beyond "classical" first-order logic. In logic programming one (usually) assumes that the program gives a *complete* description of the intended interpretation, i.e., anything which cannot be inferred from the program is assumed to be false. For example, one would say that $\neg int(a)$ is a consequence of the above program P because $int(a) \notin \mathcal{H}_P$. This means that, from a logic programming perspective, the above program captures exactly the natural numbers, something which is impossible to accomplish within first-order logic (see e.g., Corollary 4.10.1 in [27] for a formal proof).

A possible inference scheme, capturing this aspect of logic programming, was introduced in [86] and is referred to as the *closed world assumption* (CWA). The CWA cannot be expressed in first-order logic (a second-order logic axiom has to be used to that effect). Note that using the CWA leads to *non-monotonic* inferences, because the addition of new information can remove certain, previously valid, consequences. For instance, by adding the clause $int(a) \leftarrow$ to the above program the literal $\neg int(a)$ is no longer a consequence of the logic program.

Normal programs We have already touched upon the CWA. Given a formula F, this rule amounts to inferring that $\neg F$ is a logical consequence of a program P if F is not a logical consequence of P. In the context of normal programs the situation is complicated by the fact that negations can occur in the bodies of clauses and therefore the truth of $\neg F$ can propagate further and may be used to infer positive formulas as well. This entails that a normal logic program does not necessarily have a unique minimal Herbrand model. To give a meaning to normal logic programs a multitude of semantics have been developed. We cannot delve into the details of these semantics and have to refer the interested reader to, e.g., [7].

2.3 Proof theory of logic programs

We first need the following definitions:

Definition 12. **(substitution)** *A* substitution *θ is a finite set of the form $\theta = \{X_1/t_1, \ldots, X_n/t_n\}$ where $X_1, \ldots, X_n$ are distinct variables and $t_1, \ldots, t_n$ are terms such that $t_i \neq X_i$. Each element X_i/t_i of θ is called a* binding*.*

Alternate definitions of substitutions exist in the literature, but the above is the most common one in the logic programming context.

We also define an *expression* to be either a term, an atom, a literal, a conjunction, a disjunction or a program clause.

Definition 13. **(instance)** *Let $\theta = \{X_1/t_1, \ldots, X_n/t_n\}$ be a substitution and E an expression. Then the* instance *of E by θ, denoted by $E\theta$, is the expression obtained by simultaneously replacing each occurrence of a variable X_i in E by the term t.*

We present some additional useful terminology related to substitutions. If $E\theta = F$ then E is said to be *more general* than F. If E is more general than F and F is more general than E then E and F are called *variants* (of each other). If $E\theta$ is a variant of E then θ is called a *renaming substitution for E*. Because a substitution is a *set* of bindings we will denote, in contrast to, e.g., [68], the *empty* or *identity substitution* by $\emptyset$ and not by the empty sequence ϵ. Substitutions can also be applied to sets of expressions by defining $\{E_1, \ldots, E_n\}\theta = \{E_1\theta, \ldots, E_n\theta\}$.

Substitutions can also be composed in the following way:

Definition 14. **(substitution composition)** *Let $\theta = \{X_1/s_1, \ldots, X_n/s_n\}$ and $\sigma = \{Y_1/t_1, \ldots, Y_k/t_k\}$ be substitutions. Then the* composition *of θ and σ, denoted by $\theta\sigma$, is defined to be the substitution $\{X_i/s_i\sigma \mid 1 \leq i \leq n \land s_i\sigma \neq X_i\} \cup \{Y_i/t_i \mid 1 \leq i \leq k \land Y_i \notin \{X_1, \ldots, X_n\}\}$.*

When viewing substitutions as functions from expressions to expressions, then the above definition behaves just like ordinary function composition, i.e., $E(\theta\sigma) = (E\theta)\sigma$. We also have that (for proofs see [68]) the identity substitution acts as a left and right identity for composition, i.e., $\theta\emptyset = \emptyset\theta = \theta$, and that composition is associative, i.e., $(\theta\sigma)\gamma = \theta(\sigma\gamma)$.

We call a substitution θ *idempotent* iff $\theta\theta = \theta$. We also define the following notations: the set of variables occurring inside an expression E is denoted by $vars(E)$, the *domain* of a substitution θ is defined as $dom(\theta) = \{X \mid X/t \in \theta\}$ and the *range* of θ is defined as $ran(\theta) = \{Y \mid X/t \in \theta \wedge Y \in vars(t)\}$. Finally, we also define $vars(\theta) = dom(\theta) \cup ran(\theta)$ as well as the restriction $\theta|_{\mathcal{V}}$ of a substitution θ to a set of variables $\mathcal{V}$ by $\theta|_{\mathcal{V}} = \{X/t \mid X/t \in \theta \wedge X \in \mathcal{V}\}$.

The following concept will form the link between the model-theoretic semantics and the procedural semantics of logic programs.

Definition 15. (answer) *Let P be a program and $G = \leftarrow L_1, \ldots, L_n$ a goal. An* answer *for $P \cup \{G\}$ is a substitution θ such that $dom(\theta) \subseteq vars(G)$.*

Definite programs We first define correct answers in the context of definite programs and goals.

Definition 16. (correct answer) *Let P be a definite program and $G = \leftarrow A_1, \ldots, A_n$ a definite goal. An answer θ for $P \cup \{G\}$ is called a* correct answer *for $P \cup \{G\}$ iff $P \models \forall((A_1 \wedge \ldots \wedge A_n)\theta)$.*

Take for instance the program $P = \{p(a) \leftarrow\}$ and the goal $G = \leftarrow p(X)$. Then $\{X/a\}$ is a correct answer for $P \cup \{G\}$ while $\{X/c\}$ and $\emptyset$ are not.

We now present a way to calculate correct answers based on the concepts of resolution and unification.

Definition 17. (mgu) *Let S be a finite set of expressions. A substitution θ is called a* unifier *of S iff the set $S\theta$ is a singleton. θ is called* relevant *iff its variables $vars(\theta)$ all occur in S. θ is called a* most general unifier *or mgu iff for each unifier σ of S there exists a substitution γ such that $\sigma = \theta\gamma$.*

The concept of unification dates back to [45] and has been rediscovered in [87]. If a unifier for a finite set S of expressions exists then there exists an idempotent and relevant most general unifier which is unique modulo variable renaming (see [5, 68]). Unifiability of a set of expressions is decidable and there are efficient algorithms for calculating an idempotent and relevant *mgu*. See for instance the unification algorithms in [5, 68] or the more complicated but linear ones in [70, 80]. From now on we denote, for a unifiable set S of expressions, by $mgu(S)$ an idempotent and relevant unifier of S. If we just want to unify two terms t_1, t_2 then we will also sometimes write $mgu(t_1, t_2)$ instead of $mgu(\{t_1, t_2\})$.

We define the *most general instance*, of a finite set S to be the only element of $S\theta$ where $\theta = mgu(S)$. The opposite of the most general instance is the *most specific generalisation* of a finite set of expressions S, also denoted by $msg(S)$, which is the most specific expression M such that all expressions in S are instances of M. Algorithms for calculating the *msg* exist [59], and this process is also referred to as *anti-unification* or *least general generalisation*.

We can now define *SLD-resolution*, which is based on the resolution principle [87]. Its use for a programming language was first described in [56] and the name SLD (which stands for Selection rule-driven Linear resolution for Definite clauses), was coined in [9]. See e.g., [5, 68] for more details about the history.

Definition 18. (SLD-derivation step) *Let $G = \leftarrow L_1, \ldots, L_m, \ldots, L_k$ be a goal and $C = A \leftarrow B_1, \ldots, B_n$ a program clause such that $k \geq 1$ and $n \geq 0$. Then G'* is derived from G and C using θ (and L_m) *iff the following conditions hold:*

1. L_m is an atom, called the selected *atom (at position m), in G.*
2. θ is a relevant and idempotent mgu of L_m and A.
3. G' is the goal $\leftarrow (L_1, \ldots, L_{m-1}, B_1, \ldots, B_n, L_{m+1}, \ldots, L_k)\theta$.

G' is also called a resolvent *of G and C.*

In the following we define the concept of a complete SLD-derivation (we will define incomplete ones later on).

Definition 19. (complete SLD-derivation) *Let P be a definite program and G a definite goal. A* complete SLD-derivation *of $P \cup \{G\}$ is a tuple $(\mathcal{G}, \mathcal{L}, \mathcal{C}, \mathcal{S})$ consisting of a sequence of goals $\mathcal{G} = \langle G_0, G_1, \ldots \rangle$, a sequence $\mathcal{L} = \langle L_0, L_1 \ldots \rangle$ of selected literals,*[3] *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:*
- *for $i > 0$, $vars(C_i) \cap vars(G_0) = \emptyset$;*
- *for $i > j$, $vars(C_i) \cap vars(C_j) = \emptyset$;*
- *for $i \geq 0$, L_i is a positive literal in G_i and G_{i+1} is derived from G_i and C_{i+1} using θ_{i+1} and L_i;*
- *the sequences $\mathcal{G}, \mathcal{C}, \mathcal{S}$ are maximal given $\mathcal{L}$.*

The process of producing variants of program clauses of P which do not share any variable with the derivation sequence so far is called *standardising apart*. Some care has to be taken to avoid variable clashes and the ensuing technical problems; see the discussions in [53] or [32].

We now come back to the idea of a proof by refutation and its relation to SLD-resolution. In a proof by refutation one adds the negation of what is to be proven and then tries to arrive at inconsistency. The former corresponds to adding a goal $G = \leftarrow A_1, \ldots, A_n$ to a program P and the latter corresponds to searching for an SLD-derivation of $P \cup \{G\}$ which leads to $\Box$. This justifies the following definition.

Definition 20. (SLD-refutation) *An* SLD-refutation *of $P \cup \{G\}$ is a finite complete SLD-derivation of $P \cup \{G\}$ which has the empty clause $\Box$ as the last goal of the derivation.*

In addition to refutations there are (only) two other kinds of complete derivations:

- Finite derivations which do not have the empty clause as the last goal. These derivations will be called *(finitely) failed*.
- Infinite derivations. These will be called *infinitely failed*.

We can now define computed answers, which correspond to the output calculated by a logic program.

Definition 21. (computed answer) *Let P be a definite program, G a definite goal and D a SLD-refutation for $P \cup \{G\}$ with the sequence $\langle \theta_1, \ldots, \theta_n \rangle$ of mgu's. The substitution $(\theta_1 \ldots \theta_n)|_{vars(G)}$ is then called a* computed answer *for $P \cup \{G\}$ (via D).*

Theorem 1. (soundness of SLD) *Let P be a definite program and G a definite goal. Every computed answer for $P \cup \{G\}$ is a correct answer for $P \cup \{G\}$.*

Theorem 2. (completeness of SLD) *Let P be a definite program and G a definite goal. For every correct answer σ for $P \cup \{G\}$ there exists a computed answer θ for $P \cup \{G\}$ and a substitution γ such that $G\sigma = G\theta\gamma$.*

[3] Again we slightly deviate from [5, 68]: the inclusion of $\mathcal{L}$ avoids some minor technical problems wrt the maximality condition.

A proof of the previous theorem can be found in [5].

We will now examine systematic ways to search for SLD-refutations.

Definition 22. (complete SLD-tree) *A* complete SLD-tree *for $P \cup \{G\}$ is a labelled tree satisfying the following:*

1. *Each node of the tree is labelled with a definite goal along with an indication of the selected atom*
2. *The root node is labelled with G.*
3. *Let $\leftarrow A_1, \ldots, A_m, \ldots, A_k$ be the label of a node in the tree and suppose that A_m is the selected atom. Then for each clause $A \leftarrow B_1, \ldots, B_q$ in P such that A_m and A are unifiable the node has one child labelled with*
   $$\leftarrow (A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_k)\theta,$$
   where θ is an idempotent and relevant mgu of A_m and A.
4. *Nodes labelled with the empty goal have no children.*

To every branch of a complete SLD-tree there corresponds a complete SLD-derivation. The choice of the selected atom is performed by what is called a *selection rule*. Maybe the most well known selection rule is the *left-to-right selection rule* of *Prolog* [28, 95, 23], which always selects the leftmost literal in a goal. The complete SLD-derivations and SLD-trees constructed via this selection rule are called *LD-derivations* and *LD-trees*.

Usually one confounds goals and nodes (e.g., in [5, 68, 78]) although this is strictly speaking not correct because the same goal can occur several times inside the same SLD-tree.

We will often use a graphical representation of SLD-trees in which the selected atoms are identified by underlining. For instance, Fig. 1 contains a graphical representation of a complete SLD-tree for $P \cup \{\leftarrow int(s(0))\}$, where P is the program of Ex. 1.

$$\begin{tikzcd}
\leftarrow \underline{int(s(0))} \arrow[d] \\
\leftarrow \underline{int(0)} \arrow[d] \\
\square
\end{tikzcd}$$

Fig. 1. Complete SLD-tree for Example 1

Normal programs Finding an efficient proof procedure for normal programs is much less obvious than in the definite case. The most commonly used procedure is the so called *SLDNF-procedure*. It is an extension of SLD-resolution which also allows the selection of ground negative literals. Basically a selected ground negative literal $\neg A$ succeeds (with the empty computed answer $\emptyset$) if $\leftarrow A$ fails *finitely*. Similarly a selected ground negative literal fails if there exists a refutation for $\leftarrow A$. This implements what is called the "negation as failure" (NAF) rule, a less powerful but more tractable inference mechanism than the CWA.

In this course we will mainly concentrate on definite logic programs. On the rare occasions we touch upon normal programs we use the definitions of SLDNF-derivations presented in [68] based on ranks, where the rank indicates the maximal nesting of sub-derivations and sub-trees created by negative calls. Note that the definition of [68] exhibits some technical problems, in the sense that some problematic goals do not have an associated SLDNF-derivation (failed or otherwise, see [8, 7]). The definition is however sufficient for our purposes, especially since most correctness results for partial deduction (e.g., [68]), to be introduced in the next section, use this definition anyway.

Soundness of SLDNF-resolution (wrt the completion semantics) is due to Clark [22]. Unfortunately SLDNF-resolution is in general not complete, mainly (but not only) due to *floundering*, i.e., computation reaches a state in which only non-ground negative literals exist.

To remedy the incompleteness of SLDNF, several extensions have been proposed. Let us briefly mention the approach of *constructive negation* overcomes some of the incompleteness problems of SLDNF [20, 21, 34, 89, 97, 96] and can be useful inside partial deduction [44]. The main idea is to allow the selection of non-ground negative literals, replacing them by disequality constraints. For instance, given $P = \{p(a) \leftarrow\}$ the negative literal $\neg p(X)$ could be replaced by $\neg(X = a)$.

Programs with built-ins Most practical logic programs make (heavy) usage of built-ins. Although a lot of these built-ins, like e.g., `assert/1` and `retract/1`, are extra-logical and ruin the declarative nature of the underlying program, a reasonable number of them can actually be seen as syntactic sugar. Take for example the following program which uses the Prolog [28, 95, 23] built-ins $=../2$ and $call/1$.

$map(P, [], []) \leftarrow$
$map(P, [X|T], [P_X|P_T]) \leftarrow C = ..[P, X, P_X],\ call(C),\ map(P, T, P_T)$
$inv(0, 1) \leftarrow$
$inv(1, 0) \leftarrow$

For this program the query $\leftarrow map(inv, [0, 1, 0], R)$ will succeed with the computed answer $\{R/[1, 0, 1]\}$. Given that query, the Prolog program can be seen as a pure definite logic program by simply adding the following definitions (where we use the prefix notation for the predicate $=../2$):

$= ..(inv(X, Y), [inv, X, Y]) \leftarrow$
$call(inv(X, Y)) \leftarrow inv(X, Y)$

The so obtained pure logic program will succeed for $\leftarrow map(inv, [0, 1, 0], R)$ with the same computed answer $\{R/[1, 0, 1]\}$.

This means that some predicates like $map/3$, which are usually taken to be higher-order, can simply be mapped to pure definite (first-order) logic programs ([99, 77]). Some built-ins, like for instance $is/2$, have to be defined by infinite relations. Usually this poses no problems as long as, when selecting such a built-in, only a finite number of cases apply (Prolog will report a run-time error if more than one case applies while the programming language Gödel [47] will delay the selection until only one case applies).

In the remainder of this course we will usually restrict our attention to those built-ins that can be given a logical meaning by such a mapping.

3 Partial Evaluation and Partial Deduction

3.1 Partial evaluation

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. This process is illustrated in Fig. 2. The program P_S is also called the *residual program*.

The theoretical feasibility of this process, in the context of recursive functions, has already been established by Kleene [52] and is known as Kleene's S-M-N theorem. However, while Kleene was concerned with theoretical issues of computability and his construction yields specialised programs which are slower than the original, the goal of partial evaluation is to exploit the static input in order to derive more efficient programs.

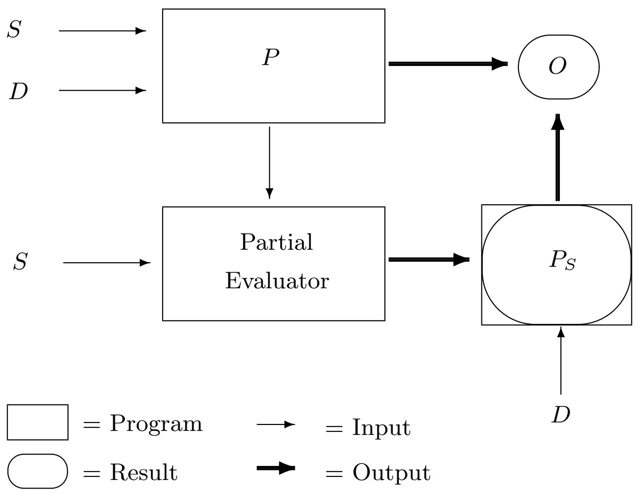


Fig. 2. Partial evaluation of programs with static and dynamic input

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. This process has therefore also been called *mixed computation* in [35]. Also, it is precisely this mixture of full evaluation steps and code generation steps (and nothing else) which distinguishes partial evaluation from other program specialisation approaches.

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. The simple example in Fig. 3 illustrates this point: the control of the loop in P is fully determined by the static input $e = 3$ and was executed beforehand by the partial evaluator, resulting in a more efficient specialised program P_e.

Partial evaluation [24, 50] has been applied to a lot of programming languages and paradigms: functional programming (e.g., [51]), logic programming (e.g., [40, 55, 81]), functional logic programming (e.g., [1, 2, 58]) term rewriting systems (e.g., [13, 14], [75]) and imperative programming (e.g., [4, 3]).

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'. In contrast to other programming languages and paradigms, one can still execute P for G' and (try to) construct a SLDNF-tree for $P \cup \{G'\}$. So, at first sight, it seems that partial evaluation for logic programs is almost trivial and just corresponds to ordinary evaluation.

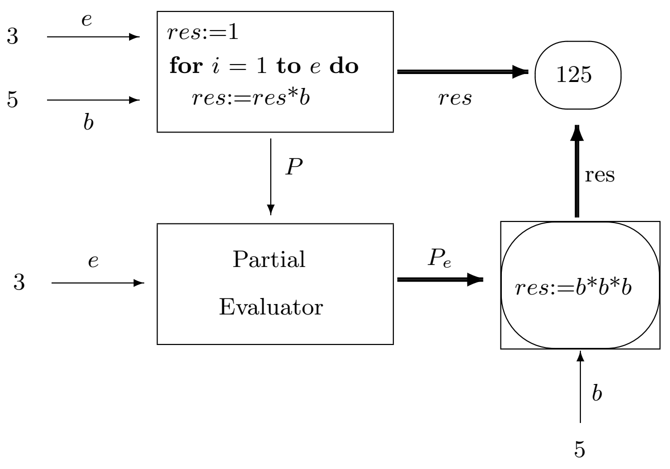


Fig. 3. Partial evaluation of a simple imperative program

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*. Its general idea is to construct a finite number of finite trees which "cover" the possibly infinite SLDNF-tree for $P \cup \{G'\}$. We will present the essentials of this technique in the next section.

The term "partial deduction" has been introduced by Komorowski (see [55]) to replace the term of partial evaluation in the context of pure logic programs. We will adhere to this terminology because the word "deduction" places emphasis on the purely logical nature of the source programs. Also, while partial evaluation of functional and imperative programs evaluates only those expressions which depend exclusively on the static input, in logic programming one can, as we have seen above, in principle also evaluate expressions which depend on the unknown dynamic input. This puts partial deduction much closer to techniques such as *supercompilation* [98, 43, 93, 90] and *unfold/fold* program transformations [19, 81], and therefore using a different denomination seems justified. We will briefly return to the relation of partial deduction to these and other techniques in the second part of this course [60] (see also [42, 49, 92]). Finally, note that program specialisation in general is not limited to just evaluating expressions, whether they depend on the static input or not. A striking illustration of this statement will be presented later in the course [60], where abstract interpretation is combined with partial deduction.

3.2 Partial deduction

In this section we present the technique of partial deduction, which originates from [54]. Other introductions to partial deduction can be found in [55, 40, 26].

In order to avoid constructing infinite SLDNF-trees for partially instantiated goals, the technique of *partial deduction* is based on constructing finite, but possibly *incomplete* SLDNF-trees. The derivation steps in these SLDNF-trees correspond to the computation steps which have already been performed 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 this section we will formalise this technique and present conditions which will ensure correctness of the so obtained specialised programs.

Definition 23. **(SLDNF-derivation)** *A SLDNF-derivation is defined like a complete SLDNF-derivation but may, in addition to leading to success or failure, also lead to a last goal where no literal has been selected for a further derivation step. Derivations of the latter kind will be called* incomplete.

An SLDNF-derivation can thus be either failed, incomplete, successful or infinite. Now, an incomplete SLDNF-tree is obtained in much in the same way.

Definition 24. *An SLDNF-tree is defined like a complete SLDNF-tree but may, in addition to success and failure leaves, also contain leaves where no literal has been selected for a further derivation step. Leaves of the latter kind are called* dangling *([72]) and SLDNF-trees containing dangling leaves are called* incomplete. *Also, an SLDNF-tree is called* trivial *iff its root is a dangling leaf, and* non-trivial *otherwise.*

The process of selecting a literal inside a dangling leaf of an incomplete SLDNF-tree and adding all the resolvents as children is called *unfolding*. An SLDNF-tree for $P \cup \{G\}$ can thus be obtained from a trivial SLDNF-tree for $P \cup \{G\}$ by performing a sequence of unfolding steps. We will return to this issue in Sect. 3.3.

Note that every branch of an SLDNF-tree has an associated (possibly incomplete) SLDNF-derivation. We also extend the notion of a *computed answer substitution (c.a.s.)* to finite incomplete SLDNF-derivations (it is just the composition of the *mgu*'s restricted to the variables of the top-level goal). Also, a *resolvent* of a finite (possibly incomplete) SLDNF-derivation is just the last goal of the derivation. Finally, if $\langle G_0, \ldots, G_n \rangle$ is the sequence of goals of a finite SLDNF-derivation, we say D has *length* n.

We will now examine how specialised clauses can be extracted from SLDNF-derivations and trees. The following definition associates a first-order formula with a finite SLDNF-derivation.

Definition 25. *Let P be a program, $\leftarrow Q$ a goal and D a finite SLDNF-derivation of $P \cup \{\leftarrow Q\}$ with computed answer θ and resolvent $\leftarrow B$. Then the formula $Q\theta \leftarrow B$ is called the* resultant *of D.*

This concept can be extended to SLDNF-trees in the following way:

Definition 26. *Let P be a program, G a goal and let τ be a finite SLDNF-tree for $P \cup \{G\}$. Let $D_1, \ldots, D_n$ be the non-failing SLDNF-derivations associated with the branches of τ. Then the set of resultants $resultants(\tau)$ is the union of the resultants of the non-failing SLDNF-derivations $D_1, \ldots, D_n$ associated with the branches of τ. We also define the set of leaves, $leaves(\tau)$, to be the atoms occurring in the resolvents of $D_1, \ldots, D_n$.*

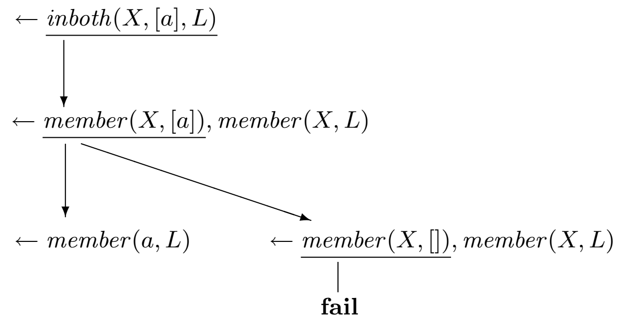
Example 2. Let P be the following program:

$member(X, [X|T]) \leftarrow$
$member(X, [Y|T]) \leftarrow member(X, T)$
$inboth(X, L1, L2) \leftarrow member(X, L1), member(X, L2)$

Figure 4 represents an incomplete SLD-tree τ for $P \cup \{\leftarrow inboth(X, [a], L)\}$. This tree has just one non-failing branch and the set of resultants $resultants(\tau)$ contains the single clause:

$$inboth(a, [a], L) \leftarrow member(a, L)$$

Note that the complete SLD-tree for $P \cup \{\leftarrow inboth(X, [a], L)\}$ is infinite.



$\leftarrow \underline{inboth(X,[a],L)}$
$\leftarrow \underline{member(X,[a])}, member(X,L)$
$\leftarrow member(a,L)$
$\leftarrow \underline{member(X,[])}, member(X,L)$
fail

Fig. 4. Incomplete SLD-tree for Example 2

If the goal in the root of a finite SLDNF-tree is atomic then the resultants associated with the tree are all clauses. We can thus formalise partial deduction in the following way.

Definition 27. (partial deduction) *Let P be a normal program and A an atom. Let τ be a finite non-trivial SLDNF-tree for $P \cup \{\leftarrow A\}$. Then the set of clauses $resultants(\tau)$ is called a* partial deduction of A in P.
If $\mathcal{A}$ is a finite set of atoms, then a partial deduction of $\mathcal{A}$ in P *is the union of one partial deduction for each element of $\mathcal{A}$.*
A partial deduction of P wrt $\mathcal{A}$ *is a normal program obtained from P by replacing the set of clauses in P, whose head contains one of the predicate symbols appearing in $\mathcal{A}$ (called the partially deduced predicates), with a partial deduction of $\mathcal{A}$ in P.*

Example 3. Let us return to the program P of Ex. 2. Based on the incomplete SLDNF-tree in Fig. 4, we can construct the following partial deduction of P wrt $\mathcal{A}$ $= \{inboth(X,[a],L)\}$:

$member(X,[X|T]) \leftarrow$
$member(X,[Y|T]) \leftarrow member(X,T)$
$inboth(a,[a],L) \leftarrow member(a,L)$

Note that if τ is a trivial SLDNF-tree for $P \cup \{\leftarrow A\}$ then $resultants(\tau)$ consists of the problematic clause $A \leftarrow A$ and the specialised program contains a loop. That is why trivial trees are not allowed in Definition 27. This is however not a sufficient condition for correctness of the specialised programs. In [69], Lloyd and Shepherdson presented and proved a fundamental correctness theorem for partial deduction. The two (additional) basic requirements for correctness of a partial deduction of P wrt $\mathcal{A}$ 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. Below we summarise the correctness result of [69].

Definition 28. (closedness, independence) *Let S be a set of first order formulas and $\mathcal{A}$ a finite set of atoms. Then S is $\mathcal{A}$-*closed *iff each atom in S, containing a predicate symbol occurring in an atom in $\mathcal{A}$, is an instance of an atom in $\mathcal{A}$. Furthermore we say that $\mathcal{A}$ is* independent *iff no pair of atoms in $\mathcal{A}$ have a common instance.*

Note that two atoms which cannot be unified may still have a common instance (i.e., unify after renaming apart). For example, $p(X)$ and $p(f(X))$ are not unifiable but have, e.g., the common instance $p(f(X))$.

Theorem 3. (correctness of partial deduction [69]) *Let P be a normal program, G a normal goal, $\mathcal{A}$ a finite, independent set of atoms, and P' a partial deduction of P wrt $\mathcal{A}$ such that $P' \cup \{G\}$ is $\mathcal{A}$-closed. Then the following hold:*

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

For instance, the partial deduction of P wrt $\mathcal{A} = \{\mathit{inboth}(X, [a], L)\}$ in Ex. 3 satisfies the conditions of Theorem 3 for the goals $\leftarrow \mathit{inboth}(X, [a], [b, a])$ and $\leftarrow \mathit{inboth}(X, [a], L)$ but not for the goal $\leftarrow \mathit{inboth}(X, [b], [b, a])$.

Note that the original unspecialised program P is also a partial deduction wrt $\mathcal{A} = \{\mathit{member}(X, L), \mathit{inboth}(X, L1, L2)\}$ which furthermore satisfies the correctness conditions of Theorem 3 for any goal G. In other words, neither Definition 27 nor the conditions of Theorem 3 ensure that any specialisation has actually been performed. Nor do they give any indication on how to construct a suitable set $\mathcal{A}$ and a suitable partial deduction wrt $\mathcal{A}$ satisfying the correctness criteria for a given goal G of interest. These are all considerations generally delegated to the *control* of partial deduction, which we discuss in the next section.

[11] also proposes an extension of Theorem 3 which uses a notion of coveredness instead of closedness. The basic idea is to restrict the attention to those parts of the specialised program P' which can be reached from G. The formalisation is as follows:

Definition 29. *Let P be a set of clauses. The* predicate dependency graph *of P is a directed graph*

- *whose nodes are the predicate symbols in the alphabet $\mathcal{A}_P$ and*
- *which contains an arc from p to q iff there exists a clause in P in which p occurs as a predicate symbol in the head and q as a predicate symbol in the body.*

Definition 30. *Let P be a program and G a goal. We say that G* depends *upon a predicate p in $\mathcal{A}_P$ iff there exists a path from a predicate symbol occurring in G to p in the predicate dependency graph of P.*
We denote by $P \downarrow_G$ the definitions in P of those predicates in $\mathcal{A}_P$ upon which G depends.
*Let $\mathcal{A}$ be a finite set of atoms. We say that $P \cup \{G\}$ is $\mathcal{A}$-*covered *iff $P \downarrow_G \cup \{G\}$ is $\mathcal{A}$-closed.*

By replacing the condition in Theorem 3 that "$P' \cup \{G\}$ is $\mathcal{A}$-closed" by the more general "$P' \cup \{G\}$ is $\mathcal{A}$-covered", we still have a valid theorem (see [11]).

Example 4. Let us again return to the program P of Ex. 2. By building a complete SLD-tree for $P \cup \{\leftarrow \mathit{member}(X, [a])\}$, we get the following partial deduction P' of P wrt $\mathcal{A} = \{\mathit{member}(X, [a])\}$:

$\mathit{member}(a, [a]) \leftarrow$
$\mathit{inboth}(X, L1, L2) \leftarrow \mathit{member}(X, L1), \mathit{member}(X, L2)$

Unfortunately, Theorem 3 cannot be applied for $G = \leftarrow \mathit{member}(X, [a])$ because $P' \cup \{G\}$ is not $\mathcal{A}$-closed (due to the body of the second clause of P'). However, $P' \cup \{G\}$ is $\mathcal{A}$-covered, because $P' \downarrow_G$ just consists of the first clause of P'. Therefore correctness of P' wrt G can be established by the above extension of Theorem 3.

The following example highlights one of the practical benefits of using partial deduction.

Example 5. Let us take the *map* program from Sect. 2.3.

$\mathit{map}(P, [], []) \leftarrow$

$$\begin{tikzcd}
\leftarrow \underline{map(inv, In, Out)} \arrow[d] \arrow[dr] & \\
\square & \leftarrow \underline{C = ..[inv, X, P_X]}, call(C), map(inv, T, P_T) \arrow[d] \\
& \leftarrow \underline{call(inv(X, PX))}, map(inv, T, P_T) \arrow[d] \\
& \leftarrow \underline{inv(X, PX)}, map(inv, T, P_T) \arrow[dl] \arrow[d] \\
\leftarrow map(inv, T, P_T) & \leftarrow map(inv, T, P_T)
\end{tikzcd}$$

Fig. 5. Unfolding Example 5

$map(P, [X|T], [P_X|P_T]) \leftarrow C = ..[P, X, P_X],\ call(C),\ map(P, T, P_T)$
$inv(0, 1) \leftarrow$
$inv(1, 0) \leftarrow$

If we now want to map the *inv* predicate on a list, then we can specialise the goal: $\leftarrow map(inv, In, Out)$. If we build the incomplete SLD-tree represented in Fig. 5 all the leaf atoms are covered and we can construct the following residual program:

$map(inv, [], []) \leftarrow$
$map(inv, [0|T], [1|P_T]) \leftarrow map(inv, T, P_T)$
$map(inv, [1|T], [0|P_T]) \leftarrow map(inv, T, P_T)$

All the higher-order overhead (i.e., the use of $= ..$ and *call*) has been removed and the function call has even been unfolded. When running the above programs (on SWI-Prolog) on a set of queries one notices that the specialised program runs about 2 times faster than the original one (and can be made even faster using filtering, as discussed in the next section).

The question that remains is, how do we come up with such (interesting and correct) partial deductions in an automatic way ? This is exactly the issue that is tackled in the next section.

3.3 Control of partial deduction

In partial deduction one usually distinguishes two levels of control [40, 74]:

- 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.

Below we examine how these two levels of control interact.

Correctness, termination and precision When controlling partial deduction the three following, often conflicting, aspects have to be reconciled:

1. *Correctness*, i.e., ensuring that Theorem 3 or its extension can be applied. This can be divided into a local condition, requiring the construction of non-trivial trees, and into a global one related to the independence and coveredness (or closedness) conditions.

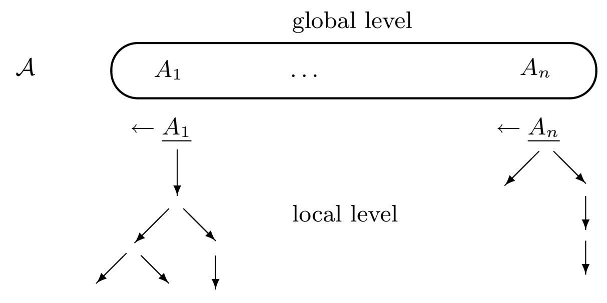


Fig. 6. Global and local level of control

2. *Termination.* This aspect 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.
3. *Precision.* For precision of the specialisation we can again discern two aspects. One which we might call *local* precision and which is related to the unfolding rule and to the fact that (potential for) specialisation can be lost if we stop unfolding an atom in $\mathcal{A}$ prematurely. Indeed, when we stop the unfolding process at a given goal Q, then all the atoms in Q are treated separately (partial deductions are defined for sets of *atoms* and not for sets of *goals*; see however "conjunctive" partial deduction which we will discuss later in this course). For instance, if we stop the unfolding process in Ex. 2 for $G = \leftarrow inboth(X, [a, b, c], [c, d, e])$ at the goal $G' = \leftarrow member(X, [a, b, c]), member(X, [c, d, e])$, partial deduction will not be able to infer that the only possible answer for G' and G is $\{X/c\}$.
   The second aspect could be called the *global* precision and is related to the structure of $\mathcal{A}$. In general having a more precise and fine grained set $\mathcal{A}$ (with more *instantiated* atoms) will lead to better specialisation. For instance, given the set $\mathcal{A} = \{member(a, [a, b]), member(c, [d])\}$, partial deduction can perform much more specialisation (i.e., detecting that the goal $\leftarrow member(a, [a, b])$ always succeeds exactly once and that $\leftarrow member(c, [d])$ fails) than given the less instantiated set $\mathcal{A}' = \{member(X, [Y|T])\}$.

A good partial deduction algorithm will ensure correctness and termination while minimising the precision loss of point 3. Let us now examine more closely how those three conflicting aspects can be reconciled.

Independence and renaming On the side of correctness there are two ways to ensure the independence condition. One is to apply a generalisation operator like the *msg* on all the atoms which are not independent (first proposed in [11]). Applying this, e.g., on the dependent set $\mathcal{A} = \{member(a, L), member(X, [b])\}$ yields the independent set $\{member(X, L)\}$. This approach also alleviates to some extent the global termination problem. However, it also diminishes the global precision and, as can be guessed from the above example, can seriously diminish the potential for specialisation.

This loss of precision can be completely avoided by using a *renaming* transformation to ensure independence. Renaming will map dependent atoms to new predicate symbols and thus generate an independent set without precision loss. For instance, the dependent set $\mathcal{A}$ above can be transformed into the independent set $\mathcal{A}' = \{member(a, L), member'(X, [b])\}$. The renaming transformation then has to map the atoms inside the residual program P' and the partial deduction goal G to the correct versions of $\mathcal{A}'$ (e.g., it has to rename the goal $G = \leftarrow member(a, [a, c]), member(b, [b])$ into $\leftarrow member(a, [a, c]), member'(b, [b])$). Renaming can often be combined with argument filtering to improve the efficiency of the specialised program. The basic idea is to filter out constants and functors and only keep the variables as arguments. For instance, instead of renaming $\mathcal{A}$ into $\mathcal{A}'$, $\mathcal{A}$ can be directly renamed into $\{mem_a(L), mem_b(X)\}$ and G into $\leftarrow mem_a([a, c]), mem_b(b)$. Further details about filtering can be found in [41], [10] or [67]. See also [84], where filtering can be obtained automatically when using folding. Filtering has also been referred to as "pushing down metaarguments" in [94] or "PDMA" in [79]. In functional programming the term of "arity raising" has also been used (and it has been studied in an offline setting, where filtering is more complicated).

Renaming and filtering are used in a lot of practical approaches (e.g., [39–41, 62, 64, 65]) and adapted correctness results can be found in [10].

Local termination and unfolding rules The local control component is usually encapsulated in what is called an unfolding rule, defined as follows.

Definition 31. *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 $P \cup \{G\}$.*

In addition to local correctness, termination and precision, the requirements on unfolding rules also include avoiding search space explosion as well as work duplication. Approaches to the local control have been based on one or more of the following elements:

- *determinacy* [41, 40, 39]
  Only (except once) select atoms that match a single clause head. The strategy can be refined with a so-called "look-ahead" to detect failure at a deeper level. Methods solely based on this heuristic, apart from not guaranteeing termination, tend not to worsen a program, but are often somewhat too conservative.
- *well-founded orders* [18, 73, 72, 71]
  Imposing some (essentially) well-founded order on selected atoms guarantees termination, but, on its own, can lead to overly eager unfolding.
- *homeomorphic embedding* [91, 65]
  Instead of well-founded ones, *well-quasi orders* can be used [12, 88]. Homeomorphic embedding on selected atoms has recently gained popularity as the basis for such an order. As shown in [61] the homeomorphic embedding relation is strictly more powerful than a large class of well-founded orders.

We will examine the above concepts in somewhat more detail. First the notion of determinate unfolding can be defined as follows.

Definition 32. **(determinate unfolding)** *A tree is* (purely) determinate *if each node of the tree has at most 1 child. An unfolding rule is* purely determinate without lookahead *if for every program P and every goal G it returns a determinate SLDNF-tree. An unfolding rule is* purely determinate (with lookahead) *if for every program P and every goal G it returns a SLDNF-tree τ such that the subtree τ^- of τ, obtained by removing the failed branches, is determinate.*

Usually the above definitions of determinate unfolding rules are extended to allow one non-determinate unfolding step, ensuring that non-trivial trees can be constructed. Depending on the definition, this non-determinate step may either occur only at the root (e.g., in [39]), anywhere in the tree or only at the bottom (i.e., its resolvents must be leaves, as in [41, 63]). These three additional forms of determinate trees, which we will call *shower*, *fork* and *beam* determinate trees respectively, are illustrated in Fig. 7.

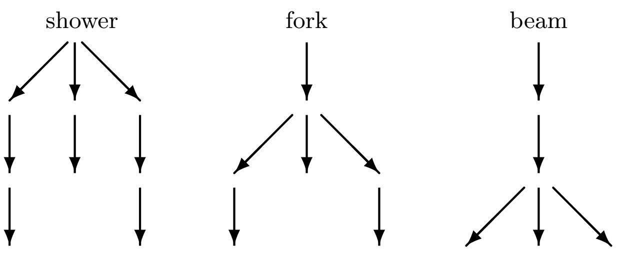


Fig. 7. Four forms of determinate trees

Determinate unfolding has been proposed as a way to ensure that partial deduction will never duplicate computations in the residual program [41, 39, 40]. Indeed, in the context of the left-to-right selection rule of Prolog, the following fairly simple example shows that non-leftmost, non-determinate unfolding may duplicate (large amounts of) work in the transformation result. The one non-determinate unfolding step performed by a shower, fork or beam determinate unfolding rule, is therefore generally supposed to mimic the runtime selection rule.

Example 6. Let us return to the program P of Ex. 2:

$member(X, [X|T]) \leftarrow$
$member(X, [Y|T]) \leftarrow member(X, T)$
$inboth(X, L1, L2) \leftarrow member(X, L1), member(X, L2)$

Let $\mathcal{A} = \{inboth(a, L1, [X, Y])\}$. By performing the non-leftmost non-determinate unfolding in Fig. 8, we obtain the following partial deduction P' of P wrt $\mathcal{A}$:

$member(X, [X|T]) \leftarrow$
$member(X, [Y|T]) \leftarrow member(X, T)$
$inboth(a, L1, [a, Y]) \leftarrow member(a, L1)$
$inboth(a, L1, [X, a]) \leftarrow member(a, L1)$

Let us examine the run-time goal $G = \leftarrow inboth(a, [z, y, \ldots, a], [X, Y])$, for which $P' \cup \{G\}$ is $\mathcal{A}$-covered. Using the Prolog left-to-right computation rule the expensive sub-goal $\leftarrow member(a, [z, y, \ldots, a])$ is only evaluated once in the original program P, while it is executed twice in the specialised program P'.

$$\begin{tikzcd}
\leftarrow \underline{inboth(a, L, [X, Y])} \arrow[d] & & \\
\leftarrow member(a, L), \underline{member(a, [X, Y])} \arrow[d] \arrow[dr] & & \\
\leftarrow member(a, L) & \leftarrow member(a, L), \underline{member(a, [Y])} \arrow[d] \arrow[dr] & \\
& \leftarrow member(a, L) & \leftarrow member(a, L), \underline{member(a, [])} \arrow[d, dash] \\
& & \textbf{fail}
\end{tikzcd}$$

Fig. 8. Non-leftmost non-determinate unfolding for Example 6

Restricting ourselves to determinate unfolding ensures that such bad cases of deterioration do not occur. It also ensures that the order of solutions, e.g., under

Prolog execution, is not altered and that termination is preserved (termination might however be improved, as e.g., $\leftarrow \mathit{loop}, \mathit{fail}$ can be transformed into $\leftarrow \mathit{fail}$; for further details related to the preservation of termination we refer to [83, 15, 17, 66]). Leftmost, non-determinate unfolding, usually allowed to compensate for the all too cautious nature of purely determinate unfolding, avoids the more drastic deterioration pitfalls in the context of, e.g., Prolog, but can still lead to multiplying unifications.

Example 7. Let us adapt Example 6 by using $\mathcal{A} = \{\mathit{inboth}(X, [Y], [V, W])\}$. We can fully unfold $\leftarrow \mathit{inboth}(X, [Y], [V, W])$ and we then obtain the following partial deduction P' of P wrt $\mathcal{A}$:

$\mathit{member}(X, [X|T]) \leftarrow$
$\mathit{member}(X, [Y|T]) \leftarrow \mathit{member}(X, T)$
$\mathit{inboth}(\underline{X}, [\underline{X}], [X, W]) \leftarrow$
$\mathit{inboth}(\underline{X}, [\underline{X}], [V, X]) \leftarrow$

No goal has been duplicated by the leftmost non-determinate unfolding, but the unification $X = Y$ for $\leftarrow \mathit{inboth}(X, [Y], [V, W])$ has potentially been duplicated. E.g., when executing the runtime goal $\leftarrow \mathit{inboth}(t_x, [t_y], [t_v, t_w])$ in P' the terms t_x and t_y will be unified when resolving with the third clause of P' and then unified again when resolving with the fourth clause of P'.[4] In the original program P this unification will only be performed once, namely when resolving with the first clause defining *member*. For run-time goals where t_x and t_y are very complicated structures this might actually result in P' being slower than the original P. However, as unifications are generally much less expensive than executing entire goals, this problem is (usually) less of an issue.

In practical implementations one has also to take care of such issues as the clause indexing performed by the compiler as well as how terms are created (i.e., avoid duplication of term construction operations). Again for these issues, determinate unfolding has proven to be a generally safe, albeit sometimes too conservative, approach. Fully adequate solutions to these, more implementation oriented, aspects are still topics of ongoing research.

Let us return to the aspect of local termination. Restricting oneself to determinate unfolding in itself does not guarantee termination, as there can be infinitely failing determinate computations. In (strict) functional programs such a condition is equivalent to an error in the original program. In logic programming the situation is somewhat different: a goal can infinitely fail (in a deterministic way) at partial deduction time but still finitely fail at run time. In applications like theorem proving, even infinite failures at run-time do not necessarily indicate an error: they might simply be due to unprovable statements. This is why, contrary to maybe functional programming, additional measures on top of determinacy should be adopted to ensure local termination.

One, albeit ad-hoc, way to solve this local termination problem is to simply impose an arbitrary depth bound. Such a depth bound is of course not motivated by any property, structural or otherwise, of the program or goal under consideration. The depth bound will therefore lead either to too little or too much unfolding in a lot of interesting cases.

As already mentioned, more refined approaches to ensure termination of unfolding exist. The methods in [18, 73, 72, 71] are based on well-founded orders, inspired by their usefulness in the context of static termination analysis (see e.g., [31, 25]). These techniques ensure termination, while at the same time allowing unfolding

[4] A very smart compiler might detect this and produce more efficient code which does not re-execute unifications.

related to the structural aspect of the program and goal to be partially deduced, e.g., permitting the consumption of static input within the atoms of $\mathcal{A}$. Formally, well-founded orders are defined as follows:

Definition 33. (wfo) *A* (strict) partial order $>_S$ *on a set* S *is an anti-reflexive, anti-symmetric and transitive binary relation on* $S \times S$. *A sequence of elements* $s_1, s_2, \ldots$ *in* S *is called* admissible wrt $>_S$ *iff* $s_i > s_{i+1}$, *for all* $i \geq 1$. *We call* $>_S$ *a* well-founded order (wfo) *iff there is no infinite admissible sequence wrt* $>_S$

To ensure local termination, one has to find a sensible well-founded order on atoms and then only allow SLDNF-trees in which the sequence of selected atoms is admissible (i.e., strictly decreasing wrt the well-founded order). If an atom that we want to select is not strictly smaller than its ancestors, we either have to select another atom or stop unfolding altogether.

Example 8. Let us return to the *member* program P of Ex. 2. A simple well-founded order on atoms of the form $member(t_1, t_2)$ might be based on comparing the list length of the second argument.
The list length $list\_length(t)$ of a term t is defined to be:
- $1 + list\_length(t')$ if $t = [h|t']$ and
- 0 otherwise.

We then define the wfo on atoms by stating $member(t_1, t_2) > member(s_1, s_2)$ iff $list\_length(t_2) > list\_length(s_2)$.

Based on that wfo, the goal $\leftarrow member(X, [a, b|T])$ can be unfolded into $\leftarrow member(X, [b|T])$ and further into $\leftarrow member(X, T)$ because the list length of the second argument strictly decreases at each step. However, $\leftarrow member(X, T)$ cannot be further unfolded into $\leftarrow member(X, T')$ because the list length does not strictly decrease.

Much more elaborate well-founded orders exist, e.g., continuously refining wfo's during the unfolding process. We refer the reader to [18, 73, 72, 71] for further details. These works also present a further refinement which, instead of requiring a decrease with every ancestor, only requires a decrease wrt the *covering ancestors*, i.e., one only compares with the ancestor atoms from which the current atom descends (via resolution).

Let us now turn our attention to approaches based on well-quasi orders, which are formally defined as follows.

Definition 34. (quasi order) *A* quasi order $\geq_S$ *on a set* S *is a reflexive and transitive binary relation on* $S \times S$.

Henceforth, we will use symbols like $<, >$ (possibly annotated by some subscript) to refer to strict partial orders and $\leq, \geq$ to refer to quasi orders. We will use either "directionality" as is convenient in the context.

Definition 35. (wbr,wqo) *Let* $\leq_S$ *be a binary relation on* $S \times S$. *A sequence of elements* $s_1, s_2, \ldots$ *in* S *is called* admissible wrt $\leq_S$ *iff there are no* $i < j$ *such that* $s_i \leq_S s_j$. *We say that* $\leq_S$ *is a* well-binary relation (wbr) *on* S *iff there are no infinite admissible sequences wrt* $\leq_S$. *If* $\leq_S$ *is a quasi order on* S *then we also say that* $\leq_S$ *is a* well-quasi order (wqo) *on* S.

Local termination is now ensured in a similar manner as for wfo's: we only allow SLDNF-trees in which the sequence of selected atoms is admissible. Observe that, while an approach based on wfo requires a strict decrease at every unfolding step, an approach based on wqo can allow incomparable steps as well. This, e.g., allows a wqo to have no a priori fixed weight or order attached to functors and arguments and means that well-quasi orders can be much more flexible and much better suited, e.g., to handle metainterpreters and metalevel encodings. See [61] for formal results substantiating that claim.

An interesting wqo is the homeomorphic embedding relation $\trianglelefteq$, which derives from results by Higman [46] and Kruskal [57]. It has been used in the context of term rewriting systems in [29, 30], and adapted for use in supercompilation [98] in [91].

The following is the definition from [91], which adapts the pure homeomorphic embedding from [30] by adding a rudimentary treatment of variables.

Definition 36. **($\trianglelefteq$)** *The* (pure) homeomorphic embedding *relation $\trianglelefteq$ on expressions is defined inductively as follows (i.e. $\trianglelefteq$ is the least relation satisfying the rules):*

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

The second rule is sometimes called the *diving* rule, and the third rule is sometimes called the *coupling* rule. When $s \trianglelefteq t$ we also say that s is *embedded in* t or t is *embedding* s. By $s \lhd t$ we denote that $s \trianglelefteq t$ and $t \not\trianglelefteq s$.

Example 9. 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 and that we even have $p(a) \lhd p(f(a))$. We also have, e.g., that:
$X \trianglelefteq X$, $p(X) \lhd p(f(Y))$, $p(X, X) \trianglelefteq p(X, Y)$ and $p(X, Y) \trianglelefteq p(X, X)$.

The homeomorphic embedding relation is very generous and will for example allow to unfold from $p([], [a])$ to $p([a], [])$ but also the other way around. This illustrates the flexibility of using well-quasi orders compared to well-founded ones, as there exists *no* wfo which will allow both these unfoldings. It however also illustrates why, when using a wqo, one has to compare with every predecessor. Otherwise one will get infinite derivations of the form $p([a], []) \rightarrow p([], [a]) \rightarrow p([a], []) \rightarrow \dots$. When using a wfo one has to compare only to the closest predecessor [72], because of the transitivity of the order and the strict decrease enforced at each step. However, wfo are usually extended to incorporate variant checking (see e.g., [71, 72]) and therefore require inspecting every predecessor anyway (though only when there is no strict weight decrease).

In order to adequately handle some built-ins, the embedding relation $\trianglelefteq$ of Definition 36 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.

Example 10. Let P be the following Prolog program:

```
le(X,X).
le(X,Y) :- X1 is X + 1, le(X1,Y).
```

If we now unfold, e.g., the goal $\leftarrow$ `le(1,0)` we get the following sequence of selected atoms, where no atom is embedding an earlier one (i.e., the sequence is admissible wrt $\trianglelefteq$): `le(1,0)` $\rightsquigarrow$ `le(2,0)` $\rightsquigarrow$ `le(3,0)` $\rightsquigarrow \infty$.

To remedy this [65], the constants and functors can be 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 [88].) The set of dynamic constants and functors is possibly infinite, and are therefore treated like the infinite set of variables in Definition 36 by adding the following rule:

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

Control of polyvariance If we use renaming to ensure independence and (for the moment) suppose that the local termination and precision problems have been solved by the approaches presented above, we are still left with the problem of ensuring *closedness* and *global termination* while minimising the *global precision loss*. We will call this combination of problems the *control of polyvariance problem* as it is very closely related to how many different specialised versions of some given predicate should be put into $\mathcal{A}$.[5] It is this important problem we address later in this course.

Let us examine how the 3 subproblems of the control of polyvariance problem interact.

- *Coveredness vs. Global Termination*
  Coveredness (or respectively closedness) can be simply ensured by repeatedly adding the uncovered (i.e not satisfying Definition 30 or Definition 28 respectively) atoms to $\mathcal{A}$ and unfolding them. Unfortunately this process generally leads to non-termination, even when using the *msg* to ensure independence. For instance, the "reverse with accumulating parameter" program (see e.g., [71, 73] or Ex. 3 in [60]) exposes this non-terminating behaviour.
- *Global Termination vs. Global Precision*
  To ensure finiteness of $\mathcal{A}$ we can repeatedly apply an "abstraction" operator which generates a set of more general atoms. Unfortunately this induces a loss of global precision.

By using the two ideas above to (try to) ensure coveredness and global termination, we can formulate a generic partial deduction algorithm. First, the concept of abstraction has to be formally defined.

Definition 37. (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 above definition guarantees that any set of clauses covered by $\mathcal{A}$ is also covered by $\mathcal{A}'$. Note that sometimes an abstraction operator is also referred to as a *generalisation operator*.

The following generic scheme, based on a similar one in [39, 40], describes the basic layout of practically all algorithms for controlling partial deduction.

Algorithm 3.1 (standard partial deduction)
Input: A program P and a goal G
Output: A specialised program P'
Initialise: $i = 0$, $\mathcal{A}_0 = \{A \mid A$ is an atom in $G\}$

```
repeat
    for each $A_k \in \mathcal{A}_i$ do
        compute a finite SLDNF-tree $\tau_k$ for $P \cup \{\leftarrow A_k\}$ by
            applying an unfolding rule $U$;
    let $\mathcal{A}'_i := \mathcal{A}_i \cup \{B_l | B_l \in leaves(\tau_k)$ for some tree $\tau_k$, such that $B_l$ is
                not an instance[6] of any $A_j \in \mathcal{A}_i\}$;
```

[5] A method is called *monovariant* if it allows only one specialised version per predicate.
[6] One can also use the variant test to make the algorithm more precise.

```
    let $\mathcal{A}_{i+1} := abstract(\mathcal{A}'_i)$; where abstract is an abstraction operator
    let $i := i + 1$;
  until $\mathcal{A}_{i+1} = \mathcal{A}_i$
  Apply a renaming transformation to $\mathcal{A}_i$ to ensure independence;
  Construct $P'$ by taking resultants.
```

In itself the use of an abstraction operator does not yet guarantee global termination. But, if the above algorithm terminates then coveredness is ensured, i.e., $P' \cup \{G\}$ is $\mathcal{A}_i$-covered (modulo renaming). With this observation we can reformulate the *control of polyvariance problem* as one of finding an *abstraction operator which maximises specialisation while ensuring termination*.

A very simple abstraction operator which ensures termination can be obtained by imposing a finite maximum number of atoms in $\mathcal{A}_i$ and using the *msg* to stick to that maximum. For example, in [73] one atom per predicate is enforced by using the *msg*. However, using the *msg* in this way can induce an even bigger *loss of precision* (compared to using the *msg* to ensure independence), because it will now also be applied on *independent* atoms. For instance, calculating the *msg* for the set of atoms $\{solve(p(a)), solve(q(f(b)))\}$ yields the atom $solve(X)$ and all potential for specialisation is probably lost.

In [73] this problem has been remedied to some extent by using a static preprocessing renaming phase (as defined in [11]) which will generate one extra renamed version for the top-level atom to be specialised. However, this technique only works well if all relevant input can be consumed in one local unfolding of this top-most atom. Apart from the fact that this huge local unfolding is not always a good idea from a point of view of efficiency (e.g., it can slow down the program as illustrated by the Examples 6 and 7), in a lot of cases this simply cannot be accomplished (for instance if partial input is not consumed but carried along, like the representation of an object-program inside a metainterpreter).

One goal pursued in the advanced part of this course [60] is to define a flexible abstraction operator which does not exhibit this dramatic loss of precision and provides a fine-grained control of polyvariance, while still guaranteeing termination of the partial deduction process.

References

1. E. Albert, M. Alpuente, M. Falaschi, P. Julián and G. Vidal. Improving Control in Functional Logic Program Specialization. In G. Levi, editor, Static Analysis. *Proceedings of SAS'98*, LNCS 1503, pages 262–277, Pisa, Italy, September 1998. Springer-Verlag.
2. M. Alpuente, M. Falaschi, and G. Vidal. Narrowing-driven partial evaluation of functional logic programs. In H. Riis Nielson, editor, *Proceedings of the 6th European Symposium on Programming, ESOP'96*, LNCS 1058, pages 45–61. Springer-Verlag, 1996.
3. L. O. Andersen. Partial evaluation of C and automatic compiler generation. In U. Kastens and P. Pfahler, editors, *4th International Conference on Compiler Construction*, LNCS 641, pages 251–257, Paderborn, Germany, 1992. Springer-Verlag.
4. L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).
5. 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.
6. K. R. Apt. *From Logic Programming to Prolog*. Prentice Hall, 1997.
7. K. R. Apt and R. N. Bol. Logic programming and negation: A survey. *The Journal of Logic Programming*, 19 & 20:9–72, May 1994.
8. K. R. Apt and H. Doets. A new definition of SLDNF-resolution. *The Journal of Logic Programming*, 8:177–190, 1994.

9. K. R. Apt and M. H. van Emden. Contributions to the theory of logic programming. *Journal of the ACM*, 29(3):841–862, 1982.
10. K. Benkerimi and P. M. Hill. Supporting transformations for the partial evaluation of logic programs. *Journal of Logic and Computation*, 3(5):469–486, October 1993.
11. K. Benkerimi and J. W. Lloyd. A partial evaluation procedure for logic programs. In S. Debray and M. Hermenegildo, editors, *Proceedings of the North American Conference on Logic Programming*, pages 343–358. MIT Press, 1990.
12. R. Bol. Loop checking in partial deduction. *The Journal of Logic Programming*, 16(1&2):25–46, 1993.
13. A. Bondorf. Towards a self-applicable partial evaluator for term rewriting systems. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 27–50. North-Holland, 1988.
14. A. Bondorf. A self-applicable partial evaluator for term rewriting systems. In J. Diaz and F. Orejas, editors, *TAPSOFT'89, Proceedings of the International Joint Conference on Theory and Practice of Software Development*, LNCS 352, pages 81–96, Barcelona, Spain, March 1989. Springer-Verlag.
15. A. Bossi and N. Cocco. Preserving universal termination through unfold/fold. In G. Levi and M. Rodriguez-Artalejo, editors, *Proceedings of the Fourth International Conference on Algebraic and Logic Programming*, LNCS 850, pages 269–286, Madrid, Spain, 1994. Springer-Verlag.
16. 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.
17. A. Bossi, N. Cocco, and S. Etalle. Transformation of left terminating programs: The reordering problem. In M. Proietti, editor, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'95*, LNCS 1048, pages 33–45, Utrecht, The Netherlands, September 1995. Springer-Verlag.
18. 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.
19. R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.
20. D. Chan. Constructive negation based on the completed database. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 111–125, Seattle, 1988. IEEE, MIT Press.
21. 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.
22. K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.
23. W. Clocksin and C. Mellish. *Programming in Prolog (Third Edition)*. Springer-Verlag, 1987.
24. 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.
25. D. De Schreye and S. Decorte. Termination of logic programs: The never ending story. *The Journal of Logic Programming*, 19 & 20:199–260, May 1994.
26. 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.
27. M. Denecker. *Knowledge Representation and Reasoning in Incomplete Logic Programming*. PhD thesis, Department of Computer Science, K.U.Leuven, 1993.
28. P. Derensart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard, Reference Manual*. Springer-Verlag, 1996.
29. N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3:69–116, 1987.
30. 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.

31. N. Dershowitz and Z. Manna. Proving termination with multiset orderings. *Communications of the ACM*, 22(8):465–476, 1979.
32. K. Doets. Levationis laus. *Journal of Logic and Computation*, 3(5):487–516, 1993.
33. K. Doets. *From Logic to Logic Programming.* MIT Press, 1994.
34. W. Drabent. What is failure ? An apporach to constructive negation. *Acta Informatica*, 32:27–59, 1995.
35. A. Ershov. Mixed computation: Potential applications and problems for study. *Theoretical Computer Science*, 18:41–67, 1982.
36. M. Fitting. *First-Order Logic and Automated Theorem Proving.* Springer-Verlag, 1990.
37. H. Fujita and K. Furukawa. A self-applicable partial evaluator and its use in incremental compilation. *New Generation Computing*, 6(2 & 3):91–118, 1988.
38. D. A. Fuller and S. Abramsky. Mixed computation of prolog programs. *New Generation Computing*, 6(2 & 3):119–141, June 1988.
39. J. Gallagher. A system for specialising logic programs. Technical Report TR-91-32, University of Bristol, November 1991.
40. 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.
41. J. Gallagher and M. Bruynooghe. The derivation of an algorithm for program specialisation. *New Generation Computing*, 9(3 & 4):305–333, 1991.
42. R. Glück and M. H. Sørensen. Partial deduction and driving are equivalent. In M. Hermenegildo and J. Penjam, editors, *Programming Language Implementation and Logic Programming. Proceedings, Proceedings of PLILP'94*, LNCS 844, pages 165–181, Madrid, Spain, 1994. Springer-Verlag.
43. 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.
44. 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.
45. J. Herbrand. Investigations in proof theory. In J. van Heijenoort, editor, *From Frege to Gödel: A Source Book in Mathematical Logic, 1879-1931*, pages 525–581. Harvard University Press, 1967.
46. G. Higman. Ordering by divisibility in abstract algebras. *Proceedings of the London Mathematical Society*, 2:326–336, 1952.
47. P. Hill and J. W. Lloyd. *The Gödel Programming Language.* MIT Press, 1994.
48. J.-M. Jacquet. *Constructing Logic Programs.* Wiley, Chichester, 1993.
49. N. D. Jones. The essence of program transformation by partial evaluation and driving. In M. S. Neil D. Jones, Masami Hagiya, editor, *Logic, Language and Computation*, LNCS 792, pages 206–224. Springer-Verlag, 1994.
50. N. D. Jones. An introduction to partial evaluation. *ACM Computing Surveys*, 28(3):480–503, September 1996.
51. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.
52. S. Kleene. *Introduction to Metamathematics.* van Nostrand, Princeton, New Jersey, 1952.
53. H.-P. Ko and M. E. Nadel. Substitution and refutation revisited. In K. Furukawa, editor, *Logic Programming: Proceedings of the Eighth International Conference*, pages 679–692. MIT Press, 1991.
54. 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.
55. J. Komorowski. An introduction to partial deduction. In A. Pettorossi, editor, *Proceedings Meta'92*, LNCS 649, pages 49–69. Springer-Verlag, 1992.
56. R. Kowalski. Predicate logic as a programming language. In *Proceedings IFIP Congress*, pages 569–574. IEEE, 1974.
57. J. B. Kruskal. Well-quasi ordering, the tree theorem, and Vazsonyi's conjecture. *Transactions of the American Mathematical Society*, 95:210–225, 1960.

58. L. Lafave and J. Gallagher. Constraint-based partial evaluation of rewriting-based functional logic programs. In N. Fuchs, editor, *Proceedings of the International Workshop on Logic Program Synthesis and Transformation (LOPSTR'97)*, LNCS 1463, Leuven, Belgium, July 1998.
59. 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.
60. M. Leuschel. *Advanced Logic Program Specialisation.* In *this volume*.
61. M. Leuschel. On the power of homeomorphic embedding for online termination. In G. Levi, editor, Static Analysis. *Proceedings of SAS'98*, LNCS 1503, pages 230–245, Pisa, Italy, September 1998. Springer-Verlag.
62. 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.
63. M. Leuschel and D. De Schreye. Constrained partial deduction and the preservation of characteristic trees. *New Generation Computing*, 16(3):283–342, 1998.
64. M. Leuschel and D. De Schreye. Creating specialised integrity checks through partial evaluation of meta-interpreters. *The Journal of Logic Programming*, 36:149–193, 1998.
65. 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.
66. M. Leuschel, B. Martens, and K. Sagonas. Preserving termination of tabled logic programs while unfolding. In N. Fuchs, editor, *Proceedings of the International Workshop on Logic Program Synthesis and Transformation (LOPSTR'97)*, LNCS 1463, Leuven, Belgium, July 1998.
67. 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.
68. J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.
69. J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11(3& 4):217–242, 1991.
70. A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, April 1982.
71. B. Martens. *On the Semantics of Meta-Programming and the Control of Partial Deduction in Logic Programming.* PhD thesis, K.U. Leuven, February 1994.
72. B. Martens and D. De Schreye. Automatic finite unfolding using well-founded measures. *The Journal of Logic Programming*, 28(2):89–146, August 1996.
73. 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.
74. 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.
75. A. Miniuissi and D. J. Sherman. Squeezing intermediate construction in equational programs. In O. Danvy, R. Glück, and P. Thiemann, editors, *Proceedings of the 1996 Dagstuhl Seminar on Partial Evaluation*, LNCS 1110, pages 284–302, Schloß Dagstuhl, 1996. Springer-Verlag.
76. T. Mogensen and A. Bondorf. Logimix: A self-applicable partial evaluator for Prolog. In K.-K. Lau and T. Clement, editors, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'92*, pages 214–227. Springer-Verlag, 1992.
77. L. Naish. Higher-order logic programming in Prolog. Technical Report 96/2, Department of Computer Science, University of Melbourne, 1995.
78. U. Nilsson and J. Małuszyński. *Logic, Programming and Prolog.* Wiley, Chichester, 1990.
79. S. Owen. Issues in the partial evaluation of meta-interpreters. In H. Abramson and M. Rogers, editors, *Meta-Programming in Logic Programming, Proceedings of the Meta88 Workshop, June 1988*, pages 319–339. MIT Press, 1989.

80. M. Paterson and M. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16(2):158–167, 1978.
81. A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *The Journal of Logic Programming*, 19& 20:261–320, May 1994.
82. 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, pages 199–213, University of Manchester, 1992. Springer-Verlag.
83. M. Proietti and A. Pettorossi. Semantics preserving transformation rules for Prolog. In *Proceedings of the ACM Symposium on Partial Evaluation and Semantics based Program Manipulation, PEPM'91*, Sigplan Notices, Vol. 26, N. 9, pages 274–284, Yale University, New Haven, U.S.A., 1991.
84. M. Proietti and A. Pettorossi. The loop absorption and the generalization strategies for the development of logic programs and partial deduction. *The Journal of Logic Programming*, 16(1 & 2):123–162, May 1993.
85. T. C. Przymusinksi. On the declarative and procedural semantics of logic programs. *Journal of Automated Reasoning*, 5(2):167–205, 1989.
86. R. Reiter. On closed world data bases. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 55–76. Plenum Press, 1978.
87. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.
88. D. Sahlin. Mixtus: An automatic partial evaluator for full Prolog. *New Generation Computing*, 12(1):7–51, 1993.
89. J. C. Shepherdson. Language and equality theory in logic programming. Technical Report PM-91-02, University of Bristol, 1991.
90. M. H. Sørensen. *Introduction to Supercompilation*. In *this volume*.
91. 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.
92. M. H. Sørensen, R. Glück, and N. D. Jones. Towards unifying partial evaluation, deforestation, supercompilation, and GPC. In D. Sannella, editor, *Programming Languages and Systems — ESOP '94. Proceedings*, LNCS 788, pages 485–500, Edinburgh, Scotland, 1994. Springer-Verlag.
93. M. H. Sørensen, R. Glück, and N. D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.
94. L. Sterling and R. D. Beer. Metainterpreters for expert system construction. *The Journal of Logic Programming*, 6(1 & 2):163–178, 1989.
95. L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, 1986.
96. P. J. Stuckey. Constructive negation for constraint logic programming. In *Proceedings, Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 328–339, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.
97. P. J. Stuckey. Negation and constraint logic programming. *Information and Computation*, 118(1):12–33, April 1995.
98. V. F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.
99. D. H. D. Warren. Higher-order extensions to Prolog: Are they needed? In J. E. Hayes, D. Michie, and Y.-H. Pao, editors, *Machine Intelligence 10*, pages 441–454. Ellis Horwood Ltd., Chicester, England, 1982.