Bottom Up Specialisation:
Towards General Foundations*

Wim Vanhoof ** Danny De Schreye*** Bern Martens †

Department of Computer Science, Katholieke Universiteit Leuven,
Celestijnenlaan 200A, B-3001, Heverlee, Belgium.
e-mail: {wimvh,dannyd,bern}@cs.kuleuven.ac.be

Abstract. In this abstract, we develop a solid theoretical foundation for a bottom-up program specialisation technique, capable of specialising a logic program w.r.t. a set of unit clauses. Extending a well-known operator, originally introduced for composing logic programs, we define a bottom-up partial deduction operator and prove correctness of the transformation with respect to the S-semantics.

1 Introduction

Partial deduction is an important transformation technique for logic programs, capable of removing inefficiencies from programs [7, 9, 4]. As an on-line specialisation technique, it is based on an evaluation mechanism for logic programs. The input to a typical partial deducer is a program and a partially instantiated query. The instantiated part represents the information with respect to which one would like to specialise; the uninstantiated part represents the information not yet known. Therefore, all classical partial deduction techniques use *top down* evaluation (or SLD-resolution) to evaluate the program parts that depend on the known input and generate a new program that computes its result using only the remainder of the input. Since the new program has less computations to perform, in general, it will be more efficient.

In recent work [12], we argued the need for a complementary partial deduction technique, capable of "specialising" a program w.r.t. a set of (unit) clauses instead of a goal. It seems natural to define such a specialisation scheme in terms of bottom-up evaluation. During evaluation, the information from the unit clauses is then propagated upwards, and new facts and clauses are derived.

* Extended Abstracts of LOPSTR'98, Eighth International Workshop on Logic-based Program Synthesis and Transformation, 15-19 June 1998, Manchester, UK. Technical Report Series, Department of Computer Science, University of Manchester, ISSN 1361-6161. Report number UMCS-98-6-1. http://www.cs.man.ac.uk/cstechrep/titles98.html

** Supported by a specialisation grant of the Flemish Institute for the Promotion of Scientific-Technological Research in Industry (IWT), Belgium.

*** Senior Research Associate of the Belgian National Fund for Scientific Research.

† Partially supported by Esprit project 25503, ARGo

In [12], we developed a specific, and very concrete control scheme for such a bottom up transformation and provided some examples showing that a combination of bottom up transformation and classical (goal-directed) top down partial deduction achieves at least equally good results as a top down scheme alone, while requiring a much less complicated control mechanism for either part.

In this work, we outline a solid theoretical foundation for such bottom up transformation schemes, and give a general framework in which concrete control strategies can be proven correct. The remainder of this abstract is organised as follows: in Section 2, we recapitulate our main motivation for a bottom-up specialisation scheme. In a subsequent section, we provide a formal framework for such bottom-up specialisation and pinpoint a condition necessary for the correctness of the transformation. Section 4 briefly describes a concrete instance of our framework and addresses the relation with compositional semantics [2, 1] and top down partial deduction. Throughout this abstract, we consider definite logic programs.

2 Motivating Bottom Up Specialisation

In this section, we recapitulate briefly our motivation for a bottom up specialisation scheme. More information can be found in [10, 12].

In a logic program, some information flows naturally in a bottom-up fashion, starting from a set of unit clauses. Take as example a program in which the concrete representation of some data structure is hidden by an abstract data type (ADT) and all operations manipulating the structure are defined using functionality provided by the ADT implementation. Abstracting such information through several layers makes sense from a software engineering point of view: concrete representations can be altered without much effort. On the other hand, every layer of abstraction decreases efficiency. Therefore, it makes sense to propagate the concrete information up into the program, to the places where it is really used. In the ADT-example: propagating the concrete representation into the program can eliminate all calls to the ADT, removing a layer of overhead.

This sort of information propagation can in principle be obtained by a top down specialisation scheme. Achieving it in a *general* and *completely automatic* way, however, is far from trivial since control information, needed by the specialiser to decide whether or not to continue the specialisation, might also flow bottom up. In particular, (Vanilla-like) meta-programs typically present difficulties of this kind [11]. Consider the following example, taken from [12]: The predicate $make\_list(T, I, R)$ can be used to create a list of a fixed length (type T), with each element initialised with I. The result is returned in R.

Example 1.

$fill\_list(L, T, I, L) \leftarrow type(T, L).$
$fill\_list(L, T, I, R) \leftarrow fill\_list([I|L], T, I, R).$
$make\_list(T, I, R) \leftarrow fill\_list([], T, I, R).$

$type(list1, [X])$
$type(list3, [X_1, X_2, X_3]).$

Example 1 represents a class of recursive predicates that build up some structure between calls before a base clause of the predicate (ending the recursion) is reached, depending on the structure built. Unfolding techniques, on which all top down specialisers are based, are known to have problems with these, since it often cannot be derived from the SLD-tree for e.g. $make\_list(list3, a, R)$ built so far whether or not unfolding such a recursive call will eventually terminate.

If this recursive predicate is handled in a bottom-up fashion, structure is shrinking between recursive calls, resulting in the facts

$make\_list(list1, I, [I]).$
$make\_list(list3, I, [I, I, I]).$

Apart from control, there are other reasons why a bottom up approach might be preferred. Sometimes goal-directed specialisation simply is not needed, because all information to be propagated into the program is already there, and need not be provided from elsewhere. Consider a program library M, in which n routines are defined, all using an ADT. A top down specialiser needs a single goal to start specialisation from, which might not be available when specialising the library. Or it/they will likely contain no information (all arguments free) since the latter flows bottom-up (further complicating top down control).

So, in a number of cases, proceeding bottom-up is a more natural solution. Bottom-up transformation and specialisation has been considered occasionally before (see e.g. [8, 3]). However, to the best of our knowledge, our ongoing effort is the first attempt to achieve these in a completely general and automatic way.

3 Defining the Framework

As mentioned in Section 1, this abstract outlines a firm foundation for bottom-up transformation. The issues of control and integration with top-down specialisation will be elaborated in future work.

First, we recall the concept of a non-ground T_P operator, acting on sets of clauses – instead of atoms – as in the context of compositional semantics (see e.g. [2, 1]). In what follows, $\mathcal{HC}$ denotes the set of all Horn clauses over a fixed first order language $\mathcal{L}$, underlying a given definite program P. Let $\equiv$ be the variant relation on $\mathcal{HC}$. For any set S, 2^S denotes its powerset.

Definition 1. *The* compositional, non-ground T_P-operator, T_P^C, *is a function* $T_P^C : 2^{\mathcal{HC}/\equiv} \to 2^{\mathcal{HC}/\equiv}$, *defined for any* $S \subseteq \mathcal{HC}/_\equiv$ *as:* $T_P^C(S) = S \cup \{H\theta \leftarrow (B_1^1, \ldots, B_{m_1}^1, \ldots, B_1^n, \ldots, B_{m_n}^n)\theta$ *where* $H \leftarrow B_1, \ldots, B_n \in P$, *and there exist* $A_1 \leftarrow B_1^1, \ldots, B_{m_1}^1, \ldots, A_n \leftarrow B_1^n, \ldots, B_{m_n}^n$ *fresh variants of clauses in* S, *such that* $\theta = mgu((B_1, \ldots, B_n), (A_1, \ldots, A_n))$ *exists.*$\}$

From [1], we know that T_P^C is continuous and its least fixpoint $T_P^C \uparrow \omega$ exists. By definition, $T_P^C \uparrow \omega$ is a set of unit clauses denoting the *least S-Herbrand model* [6, 1] of P, $M_s(P)$. That is, $M_s(P) = Heads(T_P^C \uparrow \omega)$ where *Heads* is a function that projects a set of (unit) clauses onto their heads. In the remainder of this abstract, slightly abusing notation, we simply consider $M_s(P) = T_P^C \uparrow \omega$.

Example 2. Consider the following program:

$$P : p(X,c) \leftarrow true. \qquad q(X,Y) \leftarrow p(X,Y)$$
$$p(a,a) \leftarrow true.$$

Then $M_s(P) = T_P^C \uparrow \omega \; = T_P^C \uparrow 2 = \{p(X,c), p(a,a), q(X,c), q(a,a)\}$.

Our aim is to define a partial deduction operator that transforms a program P into another program P' in which information is propagated in a bottom up fashion. T_P^C can serve as a basis for such an operator, but we need to ensure that consecutive applications of the T_P^C-operator, starting from a set $\{true \leftarrow true\}$ always terminate within finite time. To achieve this, we define and incorporate an abstraction operator on $\mathcal{HC}$.

To simplify the definitions, we introduce the following notation: Unless specified otherwise, uppercase characters A,B,... denote atoms and $\tilde{A}$,$\tilde{B}$ conjunctions of atoms. As usual, $A \preceq B$ denotes that A is more general than B.

Definition 2. *An* abstraction function*, Abst, is an idempotent function* $2^{\mathcal{HC}/\equiv} \rightarrow 2^{\mathcal{HC}/\equiv}$*, such that for any* $S \subseteq 2^{\mathcal{HC}/\equiv}$*, if* $S' = Abst(S)$*, then for every clause* $A \leftarrow \tilde{B} \in S$*, either* $A \leftarrow \tilde{B} \in S'$ *or there exists a clause* $H \leftarrow H \in S'$ *such that* $H \preceq A$*.*

An abstraction function maps a clause onto a clause with a more general head[1]. The intention is straightforward: if enough clauses generated by T_P^C can be mapped onto generalisations, instead of building the (possibly infinite) least S-Herbrand model of P, we can build a *finite* set of clauses, P', that defines the *same* least S-Herbrand model as P. Therefore, we combine T_P^C with an abstraction function to obtain a bottom-up partial deduction operator.

Definition 3. *The* abstracting T_P^C-operator*,* A_P^C*, associated to a program* P *and an abstraction function Abst is defined as* $A_P^C = Abst \circ T_P^C$*.*

Example 3. Reconsider the program of Example 2. We define $Abst(\{p(a,a) \leftarrow true.\} \cup \mathcal{C}) = \{p(a,Y) \leftarrow p(a,Y)\} \cup \mathcal{C}$ where $\mathcal{C}$ denotes any set of clauses other than $p(a,a) \leftarrow true$ (that is, only $p(a,a) \leftarrow true$ is abstracted, all other clauses remain unchanged by $Abst$). Then

$$I_0 = \{true \leftarrow true.\}$$

$$I_1 = A_P^C(I_0) = \left\{ \begin{array}{l} p(a,Y) \leftarrow p(a,Y). \\ p(X,c) \leftarrow true. \end{array} \right\} \quad I_2 = A_P^C(I_1) = \left\{ \begin{array}{l} p(a,Y) \leftarrow p(a,Y). \\ p(X,c) \leftarrow true. \\ q(a,Y) \leftarrow p(a,Y). \\ q(X,c) \leftarrow true. \end{array} \right\}$$

The reader can easily verify that the set I_2 is the least fixpoint of A_P^C, since $A_P^C(I_2) = I_2$.

In general, A_P^C is not a monotonic operator, since $Abst$ may replace clauses by their generalisations. However, the following proposition holds:

[1] Introducing a clause of the form $H \leftarrow H$ seems strange. The reasons for proceeding in this way will be clarified below.

Proposition 1. *For any $S \subseteq \mathcal{HC}/_\equiv$, such that $Abst(S) = S$, we have $S \subseteq A_P^C(S)$.*

The proof of Proposition 1 is straightforward by the definition of A_P^C and monotonicity of T_P^C. Thus, if we start off from an initial set S_0 which satisfies $Abst(S_0) = S_0$, monotonicity of a sequence of A_P^C-applications is ensured since $Abst$ is idempotent. Therefore, we can define ordinal powers of A_P^C as follows:

Definition 4. $A_P^C \uparrow 0 = \{true \leftarrow true\}$, $A_P^C \uparrow n = A_P^C(A_P^C \uparrow (n-1))$ *for* $n \in \mathbb{N}$ *and* $A_P^C \uparrow \omega = \bigcup_{n \in \mathbb{N}} A_P^C \uparrow n$.

Henceforth, we will always use A_P^C as in Definition 4. Now, when a clause $H \leftarrow H$ (with H a generalisation of an atom A) is used by T_P^C in a derivation, the resulting clauses are no longer unit clauses. Thus, instead of directly generating the elements of $M_s(P)$ derivable from A, a clause is derived that is capable of generating those elements, provided the original A is present. To that extent, we define two complementary functions on sets generated by A_P^C: $\mathcal{R}es$, containing the clauses that have been replaced by an abstraction and $\mathcal{G}en$, comprising the abstractions themselves.

Definition 5. $\mathcal{R}es, \mathcal{G}en : 2^{\mathcal{HC}/_\equiv} \rightarrow 2^{\mathcal{HC}/_\equiv}$ *are defined as:*

- $\mathcal{R}es(A_P^C \uparrow 0) = \mathcal{G}en(A_P^C \uparrow 0) = \{\}$.
- $\mathcal{R}es(A_P^C \uparrow n) = \mathcal{R}es(A_P^C \uparrow (n-1)) \cup \{C | C \in T_P^C(A_P^C \uparrow (n-1))$ *and* $C \notin A_P^C \uparrow n$ *for any* $n > 0$.
- $\mathcal{G}en(A_P^C \uparrow n) = \mathcal{G}en(A_P^C \uparrow (n-1)) \cup Abst(T_P^C(A_P^C \uparrow (n-1))) \setminus T_P^C(A_P^C \uparrow (n-1))$ *for any* $n > 0$.

Obviously, all clauses in any $\mathcal{G}en$ set are of the form $H \leftarrow H$. Note also that $\mathcal{G}en(A_P^C \uparrow n) \subseteq A_P^C \uparrow n$ while $\mathcal{R}es(A_P^C \uparrow n) \cap A_P^C \uparrow n = \{\}$.

In Example 3, $M_s(I_2) = \{p(X,c), q(X,c), p(a,c), q(a,c)\} \neq M_s(P)$. We will return to the presence of $p(a,c)$ and $q(a,c)$ in Example 4 below. Here we first note that $p(a,a)$ and $q(a,a)$ have disappeared. This is due to the abstraction of $p(a,a) \leftarrow true$ into $p(a,Y) \leftarrow p(a,Y)$. There is however a clause $q(a,Y) \leftarrow p(a,Y)$ in $A_P^C \uparrow 2$ such that $\theta = mgu(p(a,a), p(a,Y))$ exists and $q(a,Y)\theta = q(a,a)$. Hence, while $p(a,a) \leftarrow true$ is abstracted away and $q(a,a) \leftarrow true$ therefore not directly derived by A_P^C, the situation is rectified by storing $p(a,a) \leftarrow true$ through $\mathcal{R}es$ and adding it to the final program, as in Definition 6 below.

In what follows, we will assume that $Abst$ is defined such that A_P^C becomes finitary[2]; in other words, that $A_P^C \uparrow \omega = A_P^C \uparrow n_0$, for some $n_0 \in \mathbb{N}$. Note that if A_P^C is indeed finitary and reaches its least fixpoint in $n_0 \in \mathbb{N}$, $\mathcal{R}es(A_P^C \uparrow n_0)$ and $\mathcal{G}en(A_P^C \uparrow n_0)$ are necessarily finite sets, since all sets $A_P^C \uparrow n$, $n \leq n_0$ are finite.

We are now in a position to define the notion of a bottom-up partial deduction.

[2] Clearly, this is a matter of control: It involves deciding when to abstract and how to do it.

Definition 6. *Given a program P and an abstraction function Abst, giving rise to a finitary abstracting T_P^C-operator, A_P^C with least fixpoint $A_P^C \uparrow n_0$, the* bottom-up partial deduction *of P (using Abst) is the program $A_P^C \uparrow n_0 \setminus \{true \leftarrow true\} \cup \boldsymbol{\mathcal{R}es}(A_P^C \uparrow n_0)$.*

At first sight, it may seem strange that $\boldsymbol{\mathcal{G}en}$ clauses are present in P' since they are "useless" and possibly introducing non-termination in top-down evaluation. However, since our transformation is based on the T_P^C-operator, it seems natural to formulate soundness and completeness of bottom-up partial deduction in terms of least S-Herbrand models, and the presence of such clauses in P' does not influence $M_s(P')$. Moreover, as we discuss in Section 4, concrete, practical instances of the framework may need the $\boldsymbol{\mathcal{G}en}$ clauses to produce a satisfactorily transformed program.

Definition 7. *Let P' be a bottom up partial deduction of a program P. P' is* sound *w.r.t. P if $M_s(P') \subseteq M_s(P)$. P' is* complete *w.r.t. P if $M_s(P) \subseteq M_s(P')$.*

Without further measures, a bottom up partial deduction is not sound in general.

Example 4. Reconsider the program P from Example 2 and $Abst$ from Example 3. Since $A_P^C(I_2) = I_2$, the fixpoint is reached in $A_P^C \uparrow 2$ and the bottom-up partial deduction of P w.r.t. $Abst$,P', is $I_2 \cup \boldsymbol{\mathcal{R}es}(I_2) = I_2 \cup \{p(a,a) \leftarrow true.\}$ It is easy to see that, while P and P' have equal least Herbrand models, $M_s(P') \not\subseteq M_s(P)$:

$$M_s(P) = \{p(X,c), p(a,a), q(X,c), q(a,a)\}$$

$$M_s(P') = \{p(X,c), p(a,a), q(X,c), q(a,a), \boldsymbol{p(a,c)}, \boldsymbol{q(a,c)}\}.$$

In this example, due to the abstraction of $p(a,a) \leftarrow true$ into $p(a,Y) \leftarrow p(a,Y)$, we introduce a clause in P' that is capable of further instantiating elements of $M_s(P')$: $p(X,c) \in M_s(P')$ and $p(X,c)$ unifies with $p(a,Y)$ without being an instance of it, and hence $p(a,c) \in M_s(P')$ while it is not an element of $M_s(P)$. Soundness can be obtained by imposing an extra condition on a bottom up partial deduction, ensuring that if any element A of $M_s(P')$ unifies with the body atom of a clause $H \leftarrow H$ introduced by $Abst$, A must be an instance of H.

Definition 8. *Let S be a set of clauses, such that $\boldsymbol{\mathcal{G}en}(S)$ and $\boldsymbol{\mathcal{R}es}(S)$ are defined. S is* inside-closed *if for all $A \leftarrow \tilde{B} \in S \cup \boldsymbol{\mathcal{R}es}(S)$ holds: if $H \leftarrow H \in \boldsymbol{\mathcal{G}en}(S)$ such that $\theta = mgu(A,H)$ exists, then $H\theta = A$.*

This enables us to formulate the correctness of the transformation.

Theorem 1. *Let P' be a bottom up partial deduction of a program P, $P' = A_P^C \uparrow n_0 \cup \boldsymbol{\mathcal{R}es}(A_P^C \uparrow n_0)$. Then, if $A_P^C \uparrow n_0$ is inside-closed, $M_s(P') = M_s(P)$, thus P' is sound and complete w.r.t. P.*

Proof. (Outline). Using Proposition II.1 from [5], we can prove by induction that

- for all $i \leq n_0$: $M_s(P_i) \subseteq M_s(P)$ where $P_i = A_P^C \uparrow i \cup \boldsymbol{\mathcal{R}es}(A_P^C \uparrow i)$, and thus $M_s(P') \subseteq M_s(P)$.

– for every $k, T_P^C \uparrow k \subseteq M_s(P')$, and hence $M_s(P) \subseteq M_s(P')$.

A worked out proof can be found in the full version of this paper.

In [6, 1, 2] the equivalence between the operational semantics and the S-semantics is established. This important result guarantees that if $M_s(P) = M_s(P')$, the operational behavior of P and P' is essentially the same: For any goal G, $P \cup G$ has an SLD-refutation with c.a.s. θ if and only if $P' \cup G$ has an SLD-refutation with c.a.s. θ. Since all classical, top-down, partial deduction techniques are formulated with respect to this operational semantics, this equivalence result enables an integration of the two transformations.

Note that while our transformation preserves finite failures, it may convert infinite failure into finite failure, as the following example shows:

Example 5.

$P:$	$q(a) \leftarrow true.$	$P':$	$q(a) \leftarrow true.$
	$q(X) \leftarrow q(X).$		

The query $q(b)$ fails infinitely in the original program P, whereas it fails finitely in the transformed program P'.

4 Discussion

This work provides a formal framework for bottom-up specialisation. We enrich the compositional T_P^C operator [2, 1] with abstraction and thus obtain a generic bottom-up partial deduction operator A_P^C. We prove soundness and completeness of the transformation provided a particular condition is imposed on A_P^C. While at first sight, this *inside-closedness* condition seems awkward to check or impose (possibly enforcing "too general" abstractions), in a practical scheme it can completely be avoided by renaming: if $Abst$ introduces a clause $p(t_1, \ldots, t_n) \leftarrow p(t_1, \ldots, t_n)$, it is renamed into $p(t_1, \ldots, t_n) \leftarrow p'(t_1, \ldots, t_n)$. The clauses $p(s_1, \ldots, s_n) \leftarrow \tilde{B}$ that are removed by $Abst$ because the head is an instance of $p(t_1, \ldots, t_n)$, are also renamed into $p'(s_1, \ldots, s_n) \leftarrow \tilde{B}$. This renaming provides also the main motivation why clauses from $\boldsymbol{\mathcal{G}en}(A_P^C \uparrow n_0)$ are still present in P': whereas a clause of the form $p(t_1, \ldots, t_n) \leftarrow p(t_1, \ldots, t_n)$ is "useless" in P', $p(t_1, \ldots, t_n) \leftarrow p'(t_1, \ldots, t_n)$ is mandatory if we want to have the original predicate names in the atoms of $M_s(P')$.

The aim of a concrete control scheme should be to maximize the propagation of information in the program, while preserving the least S-Herbrand model. Without abstraction, the process reduces to bottom-up evaluation and $P' = M_s(P)$. In [12], such a concrete bottom up specialisation scheme is presented: an operator D_P is introduced that, starting from the unit clauses in P, builds a DAG D with atoms as nodes, and an edge from A_1 to A_2 if A_2 is an immediate consequence of A_1. Generalisation occurs by replacing a sub-DAG rooted in A with a single atom that is a generalisation of A. D is kept finite by (basically) imposing a well-quasi order on every path in D. Since no inside-closedness condition is imposed on the DAG, the residual program P' extracted from it might not be sound with respect to P: if $A \in M_s(P')$, it is possible that $A \notin M_s(P)$; however A is "correct" in the sense that there does exist an $A' \in M_s(P)$ such that $A' \preceq A$.

Enriching T_P^C with abstraction introduces several new complications. While in general A_P^C is not continuous, it can be defined finitary. This is mandatory if we plan to use it for program transformation, since the transformation as well as the resulting program must be finite. As noted in [2, 1], the result of computing the compositonal semantics of P, $T_P^C \uparrow \omega$, is in essence a set of resultants which can be seen as the result of a top down partial deduction of P w.r.t. a set of atomic goals of the form $p(X_1, \ldots, X_n)$. As a consequence of abstraction, this will not be the case with our transformation. Reconsider the program of Example 2 to which the clause $q(Y) \leftarrow p(X, Y)$ is added. *Abst* is defined as in Example 3, resulting in a set I_2 which includes the clause $q(Y) \leftarrow p(a, Y)$. Although this clause was produced by T_P^C starting from an abstraction, it can not be derived as a resultant from a top down partial deduction of $q(Y)$, since its body contains the constant a as the result of an earlier bottom-up propagation.

References

1. A. Bossi, M. Gabbrielli, G. Levi, and M. Martelli. The S-semantics approach: Theory and applications. *Journal of Logic Programming*, 19/20:149–197, 1994.
2. A. Bossi, M. Gabbrielli, G. Levi, and M. C. Meo. A compositional semantics for logic programs. *Theoretical Computer Science*, 122(1-2):3–47, 1994.
3. Y. Cosmadopoulos, M. Sergot, and R. W. Southwick. Data-driven transformation of meta-interpreters: A sketch. In H. Boley and M. M. Richter, editors, *Proceedings of the International Workshop on Processing Declarative Knowledge (PDK'91)*, volume 567 of *LNAI*, pages 301–308. Springer Verlag, 1991.
4. D. De Schreye, M. Leuschel, and B. Martens. Tutorial on program specialisation (abstract). In J.W. Lloyd, editor, *Proceedings ILPS'95*, pages 615–616, Portland, Oregon, December 1995. MIT Press.
5. F. Denis and J.P. Delahaye. Unfolding, procedural and fixpoint semantics of logic programs. In *8th Annual Symposium on Theoretical Aspects of Computer Science*, volume 480 of *LNCS*, pages 511–522. Springer, 1991.
6. M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative modeling of the operational behaviour of logic programs. *Theoretical Computer Science*, 69:289–318, 1989.
7. J. Gallagher. Specialisation of logic programs: A tutorial. In *Proceedings PEPM'93, ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–98, Copenhagen, June 1993. ACM Press.
8. H. J. Komorowski. Synthesis of programs in the partial deduction framework. In Michael R. Lowry and Robert D. McCartney, editors, *Automating Software Design*, pages 377–403. AAAI Press, 1991.
9. J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11(3&4):217–242, 1991.
10. W. Vanhoof. Bottom up information propagation for partial deduction. In M. Leuschel, editor, *Proceedings of the International Workshop on Specialization of Declarative Programs and its Applications*, pages 73 – 82, Port Jefferson, Long Island N.Y. (USA), October 1997. Proceedings available as report CW255, Dept. of Computer Science, K.U.Leuven, Belgium.

11. W. Vanhoof and B. Martens. To parse or not to parse. In N. Fuchs, editor, *Proceedings of LOPSTR'97*, Leuven, Belgium, July 1997. To appear in Lecuture Notes in Computer Science series (Springer-Verlag).
12. W. Vanhoof, B. Martens, D. De Schreye, and K. De Vlaminck. Specialising the other way around. In J. Jaffar, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, Manchester, United Kingdom, June 1998. To appear (MIT-Press).