Conjunctive Partial Deduction in Practice

Jesper Jørgensen*   Michael Leuschel**   Bern Martens***

K.U. Leuven, Department of Computer Science
Celestijnenlaan 200 A, B-3001 Heverlee, Belgium
e-mail: {jesper,michael,bern}@cs.kuleuven.ac.be

Abstract. Recently, partial deduction of logic programs has been extended to conceptually embed folding. To this end, partial deductions are no longer computed of single atoms, but rather of entire conjunctions; Hence the term "conjunctive partial deduction".
Conjunctive partial deduction aims at achieving unfold/fold-like program transformations such as tupling and deforestation within fully automated partial deduction. However, its merits greatly surpass that limited context: Also other major efficiency improvements are obtained through considerably improved side-ways information propagation. In this extended abstract, we investigate conjunctive partial deduction in practice. We describe the concrete options used in the implementation(s), look at abstraction in a practical Prolog context, include and discuss an extensive set of benchmark results. From these, we can conclude that conjunctive partial deduction indeed pays off in practice, thoroughly beating its conventional precursor on a wide range of small to medium size programs. However, controlling it in a perfect way proves far from obvious, and a range of challenging open problems remain as topics for further research.

1   Introduction

Partial deduction [27, 15, 9] is a well-known technique for automatic specialisation of logic programs. It takes as input a program and a (partially instantiated) query and returns as output a program tuned towards answering the given query and any of its instances. The process proceeds by building incomplete SLD(NF)-trees for the goal atoms, assembling specialised clauses from the leaves, and subsequently doing likewise for (new) atoms occurring in these clauses.

An important feature (in fact, a bug, one might say) lies in the fact that specialised clauses are produced for individual, separate atoms. As a consequence, partial deduction is unable to achieve typical transformations involving elimination of redundant variables [33, 34, 35], as exemplified by the query

---

$\leftarrow da(Xs, Ys, Zs, R)$ and the following program:

$$da(Xs, Ys, Zs, R) \qquad \leftarrow app(Xs, Ys, T), app(T, Zs, R).$$
$$app([\ ], Ys, Ys).$$
$$app([H\,|\,Xs], Ys, [H\,|\,Zs]) \leftarrow app(Xs, Ys, Zs).$$

The desired, but "classically" unachievable transformed program is:

$$da([\ ], Ys, Zs, R) \qquad \leftarrow app(Ys, Zs, R).$$
$$da([X\,|\,Xs], Ys, Zs, [X\,|\,Rs]) \leftarrow da(Xs, Ys, Zs, Rs).$$
$$app([\ ], Ys, Ys).$$
$$app([X\,|\,Xs], Ys, [X\,|\,Zs]) \qquad \leftarrow app(Xs, Ys, Zs).$$

To overcome such limitations, [22, 16] develop *conjunctive partial deduction*. As the name suggests, this extension of convential partial deduction no longer automatically splits up goals into constituting atoms, but attempts to specialise the program with respect to (entire) conjunctions of atoms. Sometimes, splitting a goal into subparts is still necessary to guarantee termination, but, in general, it is avoided when the latter is not the case. The necessary extensions to the basic framework [27] are elaborated in [22], while [16] discusses control aspects and [23] studies the integration with bottom-up abstract interpretation techniques. Finally, [25] develops a supporting transformation to remove remaining useless variables from programs produced by conjunctive partial deduction proper. Some essentials are (mostly) informally recapitulated below; For a formal exposé, we refer to [22, 16, 23, 25].

The resulting technique incorporates part of the unfold/fold technology [7, 39, 30], and bears some relationship to automated methods proposed in [33, 34, 35]. It also approaches more closely techniques for the specialisation and transformation of functional programs, such as deforestation [42], and supercompilation [40, 36]. Especially the latter constituted, together with unfold/fold transformations, a source of inspiration for the conception and design of conjunctive partial deduction.

In the present paper, we endeavour to put conjunctive partial deduction on trial. We use a large set of small and medium size benchmark programs taken from [20]. Together, we claim, they give a good impression of specialisation and transformation obtained by various methods on a declarative subset of Prolog. We will be particularly concerned with the resulting speedups, and also pay attention to the complexity of the transformation process itself. (In order for a method to be practically viable, it is not sufficient that it terminates "in theory"; It should actually do so within reasonable time bounds, i.e. steer clear of combinatorial explosions.) We will endeavour to make this paper as much as possible self-contained, or at least understandable in broad lines without the reader having to consult numerous other papers. However, since experiments and their results are the true subject matter of the current paper, and these experiments have been performed using methods constituted of diverse elements presented in more detail elsewhere, we will occasionally be forced to refer the reader seeking more technical details on the transformations involved to these earlier sources. We apologise beforehand if this makes the reading a bit terse for

those to whom the present paper constitutes a first introduction to conjunctive partial deduction. After all, we do want to focus on practice and experiments here, and we can only hope that the interested reader will feel encouraged to consult the respective references for more information on the various techniques lying at the basis of the work reported below.

In Section 2 then, we do briefly recapitulate core notions from "classical" as well as conjunctive partial deduction. Section 3 brushes up a well-known termination problem connected to Prolog's left-to-right (unfair) computation rule, as well as some other aspects with a slightly pedestrian, "get things going in practice" flavour, not addressed before in conjunctive partial deduction in a general logic programming setting [22, 16]. Next, Sections 4 and 5 constitute the main body of the paper, describing the particular transformation method(s) and benchmarks used, showing the results, and highlighting the most interesting aspects of the latter.

2   Background

We assume the reader to be familiar with the basic concepts of logic programming and ("conventional" or "classical" or "standard") partial deduction, as presented in [26] and [27]. Throughout the paper, we only consider definite programs and goals.

2.1   Controlling Conventional Partial Deduction

In recent years, following the foundational paper by Lloyd and Shepherdson [27], considerable progress has been achieved on the issue of controlling automated partial deduction. In that context, a clear conceptual distinction was introduced between local and global control [15, 29].

The former deals with the construction of (possibly incomplete) SLD-trees for the atoms to be partially deduced. In essence, it consists of an unfolding strategy. Requirements are: termination, good specialisation, avoiding search space explosion as well as work duplication. Approaches have been based on one or more of the following elements:

- determinacy [13, 14]
  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 measures [8, 28]
  Imposing some (essentially) well-founded order on selected atoms guarantees termination, but, on its own, can lead to overly eager unfolding.
- homeomorphic embedding [36, 24]
  Instead of well-founded ones, well-quasi-orders can be used [37, 2]. Homeomorphic embedding on selected atoms has recently gained popularity as the basis for such an order.

At the global control level, closedness [26] is ensured and the degree of poly-variance is decided: For which atoms should partial deductions be produced? Obviously, again, termination is an important issue, as well as obtaining a good overall specialisation. The following ingredients are important in recent approaches:

- characteristic trees [13, 14, 21, 19]

  A characteristic tree is an abstraction of an SLD-tree. It registers which atoms have been selected and which clauses were used for resolution. As such, it provides a good characterisation of the computation and specialisation connected with a certain atom (or goal). Its use in partial deduction lies in the control of polyvariance: Produce one specialised definition per characteristic tree encountered.

- global trees [29, 24]

  Partially deduced atoms (or characteristic atoms, see below) can be registered in a tree structure that is kept well-founded or well-quasi-ordered to ensure (global) termination. In general, doing so, while maintaining closedness, requires abstraction (generalisation).

- characteristic atoms [19, 24]

  Recent work has shown that the best control of polyvariance can be obtained not on the basis of either syntactical structure (atoms) or specialisation behaviour (characteristic trees) separately, but rather through a combination of both. Such pairs consisting of an atom and an associated (imposed) characteristic tree are called *characteristic atoms*.

Finally, subsidiary transformations, applicable in a post-processing phase, have been proposed, e.g. to remove certain superfluous structures [12, 1] or to reduce unnecessary polyvariance [24].

2.2 Conjunctive Partial Deduction

As explained in Section 1, conjunctive partial deduction has been designed with the aim of overcoming some limitations inherent in its conventional relative. The essential aspect lies in the joint treatment of entire conjunctions of atoms, connected through shared variables, at the global level (complemented, of course, with some renaming to deliver program clauses). Basically, this can be seen as a refinement of abstraction with respect to the conventional case. Indeed, in conjunctive partial deduction, a conjunction can be abstracted by either splitting it into subconjunctions, or generalising syntactic structure, or through a combination of both. See also e.g. [34] for a related generalisation operation in the context of an unfold/fold transformation technique. In classical partial deduction, on the other hand, any conjunction is always split (i.e. abstracted) into its constituent atoms before lifting the latter to the global level. Details can be found in [22, 16].

Apart from this aspect, the conventional control notions described above also apply in a conjunctive setting. Notably, the concept of characteristic atoms can be generalised to *characteristic conjunctions*, which are just pairs consisting of a conjunction and an associated characteristic tree.

3 Conjunctive Partial Deduction for Pure Prolog

We will for the remainder of the paper only be concerned with conjunctive partial deduction for pure Prolog. This means, besides disallowing non-pure features, that we suppose a static (unfair) computation rule, e.g. left-to-right, and that we will demand preservation of termination under that computation rule (in the sequel assumed "left-to-right", unless explicitly stated otherwise).

3.1 Unfolding rules

In the given context, determinate unfolding has been proposed as a way to ensure that partial deduction will never actually worsen the behaviour of the program [13, 14]. Indeed, even fairly simple examples suffice to show that non-leftmost, non-determinate unfolding may duplicate (large amounts of) work in the transformation result. Leftmost, non-determinate unfolding, usually allowed to compensate for the all too cautious nature of purely determinate unfolding, avoids the more drastic deterioration pitfalls, but can still lead to multiplying unifications.

3.2 Splitting and Abstraction

A termination problem specific to conjunctive partial deduction lies in the possible appearance of ever growing conjunctions at the global level (see Section 3 of [28] for a comparable phenomenon in the context of local control). To cope with this, abstraction [15, 24, 16] provides for the possibility of *splitting* a conjunction into several parts, thus producing *subconjunctions* of the original one. The details can be found in [16]. Let us present a simple example. Consider the two conjunctions Q_1 and Q_2:

$$Q_1 = p(X, Y) \wedge q(Y, Z)$$

$$Q_2 = p(f(X), Y) \wedge r(Z, R) \wedge q(Y, Z)$$

If specialisation of Q_1 leads to specialisation of Q_2, there is a danger of non-termination. The method proposed in [16] prevents this by first splitting Q_2 into $Q = p(f(X), Y) \wedge q(Y, Z)$ and $r(Z, R)$ and subsequently taking the msg of Q_1 and Q. As a result, only $r(Z, R)$ will be considered for further specialisation.

Now, given a left-to-right computation rule, the above operation alters the sequence in which goals are executed. Indeed, the p- and q-subgoals will henceforth be treated jointly (they will probably be renamed to a single atom). Consequently, there is no way an r-call can be interposed. From a purely declarative point of view, there is of course no reason why goals should not be interchanged, but under a fixed (unfair) computation rule, however, such *non-contiguous* splitting can worsen program performance, and even destroy termination.

In fact, the latter point has already been addressed in the context of unfold/fold transformations (see e.g. [6, 3, 5, 4]). To the best of our knowledge,

however, no satisfactory solutions, suitable to be incorporated into a fully automatic system, have yet been proposed. Below, we present an example by way of illustration for the benefit of those readers who are not yet familiar with the phenomenon from the work on unfold/fold transformations.

Consider the following program:

```
flipallint(XT,TT) :- flip(XT,TT),allint(TT).
flip(leaf(X),leaf(X)).
flip(tree(XT,Info,YT),tree(FYT,Info,FXT)) :- flip(XT,FXT), flip(YT,FYT).
allint(leaf(X)) :- int(X).
allint(tree(L,Info,R)) :- int(Info), allint(L), allint(R).
int(0).
int(s(X)) :- int(X).
```

The deforested version, obtained by conjunctive partial deduction using the control of [16], would be:

```
flipallint(leaf(X),leaf(X)) :- int(X).
flipallint(tree(XT,Info,YT),tree(FYT,Info,FXT)) :-
        int(Info), flipallint(XT,FXT), flipallint(YT,FYT).
```

where the transformed version of int is unchanged. Under a left-to-right computation rule, the query flipallint(tree(leaf(Z),0,leaf(a)),Res) terminates with the original, but not with the deforested program.

Contiguous Splitting For this reason, in the benchmarks below, we have in all but two cases *limited splitting to be contiguous*, that is, we split into contiguous subconjunctions only. (This can be compared with the outruling of goal switching in [3].) As a consequence, compared to the basic (declarative) method in [16], on the one hand, some opportunities for fruitful program transformation are left unexploited, but, on the other hand, Prolog programs are significantly less prone to actual deterioration rather than optimisation. There are many variations on how to define contiguous subconjunctions. In [16], a non-contiguous method for splitting was presented, based on *maximal connected subconjunctions*[4]:

Definition 1. (maximal connected subconjunctions) Given a conjunction $Q \equiv A_1 \wedge \ldots \wedge A_n$[5], the collection mcs$(Q) = \{Q_1, \ldots, Q_m\}$ of *maximal connected subconjunctions* is defined through the following conditions:

1. $Q = Q_1 \wedge \ldots \wedge Q_m$
2. If a variable X occurs in both A_i and A_j where $i < j$, then A_i occurs before A_j in the same Q_k.

---

[4] This notion is closely related to those of "variable-chained sequence" and "block" of atoms used in [33, 34].

[5] In this definition and the remainder of this paper, = is modulo reordering and ≡ is not.

We can define the basic notion of maximal contiguous connected subconjunctions in a similar way:

Definition 2. (maximal contiguous connected subconjunctions) For a given conjunction $Q \equiv A_1 \wedge \ldots \wedge A_n$, the collection $\text{mccs}(Q) = \{Q_1, \ldots, Q_m\}$ of *maximal contiguous connected subconjunctions* is defined through the following conditions:

1. $Q \equiv Q_1 \wedge \ldots \wedge Q_m$
2. $\text{mcs}(Q_i) = \{Q_i\}$ for all $i \leq m$
3. $Vars(Q_i) \cap Vars(Q_{i+1}) = \emptyset$ for all $i < m$

The conjunctions $p(X) \wedge p(Y) \wedge q(X, Y)$, $r(Z, T)$ and $p(Y)$ are the maximal contiguous connected subconjunctions (mccs) of $p(X) \wedge p(Y) \wedge q(X, Y) \wedge r(Z, T) \wedge p(Y)$. Other definitions of contiguous subconjunctions could disallow built-ins and/or negative literals in the subconjunctions or allow unconnected atoms inside the subconjunctions, e.g. like $p(S)$ in $p(X) \wedge p(S) \wedge q(X, Y)$.

Static conjunctions Actually, the global control regime used in some of our experiments deviates from the one described by [16] in one further aspect. Even though abstraction (splitting) ensures that the length of conjunctions (the number of its atoms) remains finite, there are (realistic) examples where the length gets very large. This, combined with the use of homeomorphic embeddings (or lexicographical orderings for that matter), leads to very large global trees, large residual programs and a bad transformation time complexity.

Take for example global trees just containing atomic goals with predicates of arity k and having as argument just ground terms $s(s(\ldots s(0)\ldots))$ representing the natural numbers up to a limit n. Then we can construct branches in the global tree having as length $l = (n + 1)^k$. Indeed for $n = 1, k = 2$ we can construct a branch of length $2^2 = 4$: $p(s(0), s(0)), p(s(0), 0), p(0, s(0)), p(0, 0)$ while respecting homeomorphic embedding or lexicographical ordering.[6]

When going to conjunctive partial deduction the number of argument positions k is no longer bounded, meaning that, even when the terms are restricted to some natural depth, the size of the global tree can be arbitrarily large. Such a kind of explosion can actually occur for realistic examples, notably for meta-interpreters written in the ground representation specialised for partially known queries (see the benchmarks).

One way to ensure that this does not happen is to limit the conjunctions that may occur at the global level. For this we have introduced the notion of *static conjunctions*. A static conjunction is any conjunction that can be obtained by *non-recursive* unfolding of the goal to be partially evaluated (or a generalisation thereof). The idea is then, by a static analysis, to compute a set of static conjunctions S from the program and the goal, and then during partial deduction

---

[6] Of course, by not restricting oneself to natural numbers up to a limit n we can construct arbitrarily large branches starting from the same $p(s(0), s(0))$: $p(s(0), s(0)), p(s(s(\ldots s(0)\ldots))), 0), p(s(s(\ldots 0\ldots))), 0), \ldots$.

only to allow conjunctions (at the global level) that are abstracted by one of the elements of $\mathcal{S}$. This is ensured by further splitting of the disallowed conjunctions. (A related technique is used in [34].) In our implementation, we use a very simpleminded way of approximating the set of static conjunctions, based on counting the maximum number of occurences of each predicate symbol in a conjunction in the program or in the goal to be partially deduced. Then $\mathcal{S}$ approximates all conjunctions where each predicate occurs at most as many times as specified by its associated maximum. In the example above, the maximum for `flip` and `allint` is 2, while for the other predicates it is 1.

Another approach, investigated in the experiments, is to avoid using homeomorphic embeddings on conjunctions, but go to a less explosive strategy, e.g. requiring a decrease in the total term size.

4   The System and the Implemented Methods

The partial evaluation system we used is called ECCE and is developed by Leuschel [20]. The system consists of a generic algorithm to which one may add one's own methods for unfolding, partitioning, abstraction, etc. All built-ins handled by the system are supposed to be declarative (e.g. ground is supposed to be delayed until ground,...). Some of the built-ins that are handled are: $=$, is, $<$, $=<$, $<$, $>=$, nonvar, ground, number, atomic, call, $\backslash==$, $\backslash=$. In the following we will give a short description of the different methods that we used in the experiments.

4.1   The Algorithm

The system implements a variant of the concrete algorithm described in [16]. The algorithm uses a global tree γ with nodes labeled with (characteristic) conjunctions. When a conjunction Q gets unfolded, then the conjunctions in the bodies of the resultants of Q (maybe further split by the abstraction) are added as child nodes (leaves) of Q in the global tree.

Algorithm 1

Input: a program P and a goal $\leftarrow Q$
Output: a set of conjunctions $\mathcal{Q}$
Initialisation: $i := 0$; $\gamma_0 :=$ the global tree with a single node, labeled Q
repeat
    1. for all leaves L in γ_i labeled with conjunction Q_L and for all bodies B in $U(P, Q_L)$ do:
        (a) $\mathcal{Q} = partition(B)$
        (b) for all Q_i in $\mathcal{Q}$ do:
            i. remove Q_i from $\mathcal{Q}$
            ii. **if** $whistle(\gamma_i, L, Q_i)$ **then** $\mathcal{Q} = \mathcal{Q} \cup abstract(\gamma_i, L, Q_i)$
                elseif Q_i is not an instance of a node in γ_i **then** add a child L' to L with label Q_i
    2. $i := i + 1$
until $\gamma_i = \gamma_{i-1}$
output the set of nodes in γ_i

The function U does the local unfolding. It takes a program and a conjunction and produces a set of (generalised) resultants: $Q \leftarrow Q'$. The function *partition* does the initial splitting of the bodies into maximal contiguous connected subconjunctions (or mcs's or plain atoms for standard partial deduction). Then for each of the subconjunctions it is checked if there is a risk of non-termination. This is done by the function *whistle*. The whistle will look at the labels (conjunctions) on the branch in the global tree to which the new conjunction Q_i is going to be added as a child and if Q_i is "larger" than one of these, it returns true. Finally, if the "whistle blows" for some subconjunction Q_i, then Q_i is abstracted by using the function *abstract*. After the algorithm terminates the residual program is obtained from the output by unfolding and renaming (details can be found in [22, 16, 25]).

Concrete Settings We have concentrated on four local unfolding rules:

1. safe determinate (t-det.): do determinate unfolding allowing one left-most non-determinate step using homeomorphic embedding with covering ancestors of selected atoms to ensure finiteness.
2. safe determinate indexed unfolding (l-idx). The difference with t-det. is that more than one left-most non-determinate unfolding step is allowed. However only "indexed" unfolding is then allowed, i.e. it is ensured that the unification work that might get duplicated is captured by the Prolog indexing (which may depend on the particular compiler). Again, homeomorphic embeddings are used to ensure finiteness.
3. homeomorphic embedding and reduction of search space (h-rs): non-left-most unfolding is allowed if the search space is reduced by the unfolding. In other words, an atom $p(\bar{t})$ can be selected if it does not match all the clauses defining p. Again, homeomorphic embeddings are used to ensure finiteness. Note that, in contrast to 2 and 3, this method might worsen the backtracking behaviour.
4. "Mixtus"-like unfolding (x): See [37] for further details (we used $max\_rec = 2$, $max\_depth = 2$, $max finite = 7$, $maxnondeterm = 10$ and only allowed non-determinate unfolding when no user predicates were to the left of the selected literal).

The measures that we have used in whistles are the following:

1. homeomorphic embedding (homeo.) on the conjunctions
2. termsize on the conjunctions
3. homeomorphic embedding (homeo.) on the conjunctions and homeomorphic embedding on the associated characteristic trees
4. termsize on the conjunctions and homeomorphic embedding on the characteristic trees

Abstraction is always done by possibly splitting conjunctions further and then taking the msg as explained in Subsection 3.2. One method (SE-hh-x) also uses the ecological partial deduction principle [19] to ensure preservation of characteristic trees upon generalisation (something which we have not yet

implemented for the conjunctive methods). The methods we have used for partitioning are based either on splitting into mcs's (non-contiguous) or into maximal contiguous connected subconjunctions. Additionally we may limit the size of conjunctions by using static conjunctions.

All unfolding rules were complemented by a simple more specific transformation in the style of SP [14] and allow the selection of ground negative literals. Post-processing removal of unnecessary polyvariance [24], determinate post-unfolding as well as redundant argument filtering [25] were always enabled.

A further extension wrt [19, 24] relates to built-ins which are also registered in the characteristic tree. The only problematic aspect is that, when generalising built-ins which generate bindings (like is/2, =../2) and which are no longer executable after generalisation, these built-ins have to be removed from the generalised characteristic tree (i.e. they are no longer selected).

5 Benchmarks

For the experimentation, we have adopted a practical approach and measured what a normal user sees. In particular, we do not count the number of inferences (the cost of which varies a lot) or some other abstract measure, but the actual execution time and size of compiled code (using Prolog by BIM 4.0.12).

The benchmark programs are taken from [20]; Short descriptions are given in Appendix A. Tables showing the results of the experiments, as well as further details, can be found in Appendix B. The results are summarised in Tables 1 and 2. We also compared to the existing systems MIXTUS [37], PADDY [32] and SP [14, 15]. In Table 1, transformation times (TT) of ECCE and MIXTUS also include time to write to file. Time for SP does not and for PADDY we do not know. The ∞ means abnormal termination (user interrupt, crash or heap overflow) occurred for some examples. $> 12h$, on the other hand, signifies that the execution had not terminated after 12 hours (see Appendix B). The unfolding used by SP does not seem to be simply determinate unfolding (look e.g. at the results for *depth.lam*), hence the "?" in Table 1.

5.1 Analysing the Results

One conclusion of the experiments is that conjunctive partial deduction (using determinate unfolding and contiguous splitting) pays off while guaranteeing no (serious) slowdown. In fact, the cases where there is a slowdown are some of those that were designed to show the effect of deforestation (`flip`, `match-append`, `maxlength` and `upto.sum2`). Two of these are handled well by the methods using non-contiguous splitting. On the fully unfoldable benchmarks, S-hh-t gave a speedup of 2.57 while Csc-hh-t achieved a speedup of 5.90, illustrating nicely that conjunctive partial deduction diminishes the need for agressive unfolding. Notice that Mixtus and Paddy have very agressive unfolding rules and fare well on the fully unfoldable benchmarks. However, on the non-fully unfoldable ones, even S-hh-t, based on determinate unfolding, is already better. The best standard

System	Partition			Whistle		Unf	Total	Total
	C/S	S/D	Contig	Conj	Chtree		Speedup	TT (min)
Cdc-hh-t	Conj	dyn.	contig	homeo	homeo	t-det	1.93	62.46
Csc-hh-t	Conj	static	contig	homeo	homeo	t-det	1.89	29.72
Csc-th-t	Conj	static	contig	termsize	homeo	t-det	1.92	5.95
Csc-hn-t	Conj	static	contig	homeo	none	t-det	1.89	35.49
Csc-tn-t	Conj	static	contig	termsize	none	t-det	1.76	2.67
Cdc-th-t	Conj	dyn.	contig	termsize	homeo	t-det	1.96	31.18
Csc-th-li	Conj	static	contig	termsize	homeo	l-idx	1.89	$> 12h + 12.95$
Cdm-hh-t	Conj	dyn.	mcs	homeo	homeo	t-det	2.00	$> 12h + 110.49$
Csm-hh-h	Conj	static	mcs	homeo	homeo	h-rs	0.77	$> 12h + 73.55$
S-hh-t	Std	-	-	homeo	homeo	t-det	1.56	3.00
S-hh-li	Std	-	-	homeo	homeo	l-idx	1.65	14.95
SE-hh-x	Std-Eco	-	-	homeo	homeo	mixtus	1.76	2.96
Mixtus	Std	-	-	mixtus	none	mixtus	1.65	$\infty + 2.71$
Paddy	Std	-	-	mixtus	none	mixtus	1.65	$\infty + 0.31$
SP	Std	-	-	pred =	=	det ?	1.34	$3*\infty + 1.99$

Table 1. Overview of all methods

System	Total Speedup	Weighted Speedup	Fully Unfoldable Speedup	Not Fully Unfoldable Speedup	Average Relative Size (orig = 1)
Cdc-hh-t	1.93	2.44	5.90	1.66	2.39
Csc-hh-t	1.89	2.38	5.90	1.62	2.02
Csc-th-t	1.92	2.44	5.90	1.65	1.68
Csc-hn-t	1.89	2.40	5.90	1.62	1.67
Csc-tn-t	1.76	2.18	4.48	1.54	1.53
Cdc-th-t	1.96	<u>2.49</u>	5.90	1.69	2.27
Csc-th-li	1.89	2.38	7.07	1.61	1.79
Cdm-hh-t	<u>2.00</u>	2.39	5.90	<u>1.72</u>	3.17
Csm-hh-h	0.77	0.52	6.16	0.63	3.91
S-hh-t	1.56	1.86	2.57	1.42	1.60
S-hh-li	1.65	2.09	4.88	1.42	1.61
SE-hh-x	1.76	2.24	<u>8.36</u>	1.48	1.46
Mixtus	1.65	2.11	8.13	1.38	1.67
Paddy	1.65	2.00	8.12	1.38	2.49
SP	1.34	1.54	2.08	1.23	<u>1.18</u>

Table 2. Short summary of the results (higher speedup and lower code size is better, the notion "weighted" is explained in Appendix B)

11

partial deduction method, for both runtime and (apart from SP) code size, is SE-hh-x. Still, compared to any of the standard partial deduction methods, our conjunctive methods (except for Csm-hh-h, Csc-tn-t, which are not meant to be competitors anyway) have a significantly better average speedup.

Furthermore, the experiments also show that the process of performing conjunctive partial deduction can be made efficient, especially if one uses determinate unfolding combined with a termsize measure on conjunctions (Csc-th-t and Csc-tn-t) in which case the average transformation time is comparable with that of standard partial deduction. Of course only further experiments may show how the transformation times grow whith the size of programs. In fact, the system was not written with efficiency as a first concern and there is a lot of room for improvement on this point.

Next, the experiments demonstrate that using the termsize measure instead of homeomorphic embedding on conjunctions clearly improves the average transformation time without loosing too much specialisation. But they also show that if one uses the termsize measure, then the use of characteristic trees becomes vital (compare Csc-th-t and Csc-tn-t). However, methods with homeomorphic embedding on conjunctions (e.g. Csc-hn-t), do not seem to benefit from adding homeomorphic embedding on characteristic trees as well (e.g. Csc-hh-t). This, at first sight somewhat surprising phenomenon, can be explained by the facts that, for the benchmarks at hand, the homeomorphic embedding on conjunctions, in a global tree setting, is already a very generous whistle, and, in the absence of negation (see the discussions in [24]), a growing of the conjunction will often result in a growing of the characteristic tree as well. Even more surprisingly, on the other hand, nevertheless adding embedding checks on characteristic trees does *not* result in larger overall transformation times. Quite on the contrary: it even reduces them!

Comparing Csc-hh-t and Cdc-hh-t, one can see that using static conjunctions also pays off in terms of faster transformation time without much loss of specialisation. If one looks more closely at the results for the two methods, then the speedup and the transformation times are more or less the same for the two methods except for the rather few cases where static conjunctions were needed: `groundunify.complex`, `liftsolve.db2`, `regexp.r2`, `regexp.r3`, `remove2` and `imperative.power`. For those cases, the loss of speedup due to the use of static conjunctions was small or insignificant while the improvement in transformation time was considerable.

Comparing Csc-th-li to Csc-th-t, one sees that indexed unfolding does not seem to have a definite effect for conjunctive partial deduction. In some case the speedup is better and in some cases worse. Only for `relative.lam` is indexed unfolding much better than determinate, but this corresponds to a case where the program can be completely unfolded. This is of course partially due to the fact that conjunctive partial deduction diminishes the need for aggressive unfolding, but for some examples it would still be highly beneficial to allow more than "just" determinate unfolding. For standard partial deduction however, indexed (as well as "Mixtus"-like) unfolding leads to definite improvement over

determinate unfolding. Note that the "Mixtus"-like unfolding used by SE-hh-x does not seem to pay off for conjunctive partial deduction at all. In a preliminary experiment, the method Csc-th-x only produced a total speedup of 1.69, i.e. only slightly better than MIXTUS or PADDY and worse than SE-hh-x. In future work we will examine how more aggressive unfolding rules can be more sucessfully used for conjunctive partial deduction.

For some benchmarks, the best speedup is obtained by the non-safe methods Cdm-hh-t or Csm-hh-h based on non-contiguous mcs splitting. But one can also see that these methods may in some cases lead to a considerable slowdown (`missionaries` and `remove`) and sometimes even to errors (`imperative.power` and `upto.sum1`) because the point at which built-ins are evaluated has been changed. This shows that methods based on non-contiguous splitting can lead to better specialisation due to *tupling* and *deforestation*, but that we need some method to control the splitting and unfolding to ensure that no slowdown, or change in termination can occur.

Conclusion

It looks like conjunctive partial deduction is efficient and pays off compared to standard partial deduction, but there are still many unsolved problems. Indeed, the speedups compared to standard partial deduction are significant but less dramatic than we initially expected. This is due to the fact that non-contiguous conjunctive partial deduction on the one hand often leads to serious slowdowns and is not really practical for most applications, while contiguous conjunctive partial deduction on the other hand is in general too weak to deforest or tuple datastructures.

Therefore it is vital, if one wants to more heavily exploit the advantages of conjunctive partial deduction, to add non-contiguous splitting (i.e. reordering) in a safe way which guarantees no serious slowdown. A first step towards a solution is presented in [5], but it remains quite restrictive and considers only ground queries. Another, more pragmatic approach might be based on making use of some mode system to allow reordering of literals as long as the resulting conjunction remains well-moded. This would be very similar to the way in which the compiler for Mercury [38] reorders literals to create different modes for the same predicate. For the semantics of Mercury any well-moded re-ordering of the literals is allowed. Altough this approach does not ensure the preservation of termination, it is then simply considered a programming error if one well-moded query terminates while the other does not.

Acknowledgements

Dan Sahlin and Zoltan Somogyi, as well as interesting remarks by anonymous referees.

References

1. K. Benkerimi and P. M. Hill. Supporting transformations for the partial evaluation of logic programs. *Journal of Logic and Computation*, 3(5):469–486, 1993.
2. R. Bol. Loop checking in partial deduction. *Journal of Logic Programming*, 16:25–46, 1993.
3. A. Bossi and N. Cocco. Preserving Universal Termination through Unfold/Fold. In G. Levi and M. Rodriguez-Artalejo, editors, *Proc. 4th International Conference on Algebraic and Logic Programming*, Lecture Notes in Computer Science 850, pages 269–286, Madrid, Spain, 1994. Springer-Verlag.
4. A. Bossi and N. Cocco. Replacement can Preserve Termination. *In this Volume.*
5. 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*, Lecture Notes in Computer Science 1048, pages 33–45, Utrecht, Netherlands, September 1995. Springer-Verlag.
6. A. Bossi and S. Etalle. Transforming Acyclic Programs. *Transactions on Programming Languages and Systems*, 16(4):1081–1096,1994.
7. R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.
8. 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.
9. D. De Schreye, M. Leuschel, and B. Martens. Program specialisation for logic programs. Tutorial. Abstract in J. Lloyd, editor, *Proceedings ILPS'95*, pages 615–616, Portland, Oregon, December 1995, MIT Press.
10. D. A. de Waal and J. Gallagher. Specialisation of a Unification Algorithm. In K.-K. Lau and T. Clement, editors, *Proceedings LOPSTR'91*, pages 205–220, Springer-Verlag, 1993.
11. D. A. de Waal and J. Gallagher. The applicability of logic program analysis and transformation to theorem proving. In A. Bundy, editor, *Proceedings CADE-12*, pages 207–221, Nancy, France, June/July 1994. Springer-Verlag, LNAI 814.
12. J. Gallagher and M. Bruynooghe. Some low-level source transformations for logic programs. In M. Bruynooghe, editor, *Proceedings Meta'90*, pages 229–244, Leuven, April 1990.
13. J. Gallagher and M. Bruynooghe. The derivation of an algorithm for program specialisation. *New Generation Computing*, 9(3 & 4):305–333, 1991.
14. J. Gallagher. A system for specialising logic programs. Technical Report TR-91-32, University of Bristol, November 1991.
15. J. Gallagher. Tutorial on specialisation of logic programs. In *Proceedings PEPM'93*, pages 88–98. ACM Press, 1993.
16. R. Glück, J. Jørgensen, B. Martens, and M.H. Sørensen. Controlling conjunctive partial deduction of definite logic programs. In H. Kuchen and S.D. Swierstra, editors, *Proceedings of the International Symposium on Programming Languages: Implementations, Logics and Programs (PLILP'96)*, LNCS 1140, pages 152–166, Aachen, Germany, September 1996. Extended version as Technical Report CW 226, K.U. Leuven. Accessible via http://www.cs.kuleuven.ac.be/~lpai.

17. T. Horváth. Experiments in partial deduction. Master's thesis, Departement Computerwetenschappen, K.U.Leuven, Leuven, Belgium, July 1993.

18. J. Lam and A. Kusalik. A comparative analysis of partial deductors for pure Prolog. Technical report, Department of Computational Science, University of Saskatchewan, Canada, May 1990. Revised April 1991.

19. M. Leuschel. Ecological partial deduction: Preserving characteristic trees without constraints. In M. Proietti, editor, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'95*, Lecture Notes in Computer Science 1048, pages 1–16, Utrecht, Netherlands, September 1995. Springer-Verlag.

20. M. Leuschel. The ECCE partial deduction system and the DPPD library of benchmarks. Accessible via `http://www.cs.kuleuven.ac.be/~lpai`.

21. M. Leuschel and D. De Schreye. An almost perfect abstraction operation for partial deduction using characteristic trees. Technical Report CW 215, Departement Computerwetenschappen, K.U. Leuven, Belgium, October 1995. Submitted for Publication. Accessible via `http://www.cs.kuleuven.ac.be/~lpai`.

22. M. Leuschel, D. De Schreye, and A. de Waal. A conceptual embedding of folding into partial deduction: Towards a maximal integration. In Michael Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming JICSLP'96*, pages 319–332, Bonn, Germany, September 1996. MIT Press. Extended version as Technical Report CW 225, K.U. Leuven. Accessible via `http://www.cs.kuleuven.ac.be/~lpai`.

23. M. Leuschel and D. De Schreye. Logic program specialisation: How to be more specific. In H. Kuchen and S.D. Swierstra, editors, *Proceedings of the International Symposium on Programming Languages: Implementations, Logics and Programs (PLILP'96)*, LNCS 1140, pages 137–151, Aachen, Germany, September 1996. Extended version as Technical Report CW 232, K.U. Leuven. Accessible via `http://www.cs.kuleuven.ac.be/~lpai`.

24. M. Leuschel and B. Martens. Global control for partial deduction through characteristic atoms and global trees. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Proceedings of the 1996 Dagstuhl Seminar on Partial Evaluation*, LNCS 1110, pages 263–283, Schloß Dagstuhl, 1996. Extended version as Technical Report CW 220, K.U. Leuven. Accessible via `http://www.cs.kuleuven.ac.be/~lpai`.

25. M. Leuschel and M.H. Sørensen. Redundant argument filtering of logic programs. *In this Volume*.

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

27. J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11:217–242, 1991.

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

29. 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. Extended version as Technical Report CSTR-94-16, University of Bristol.

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

31. D. L. Poole and R. Goebel. Gracefully adding negation and disjunction to Prolog. In E. Shapiro, editor, *Proceedings ICLP'86*, pages 635–641, London, U.K., July

15

1986. Springer-Verlag, LNCS 225.

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

33. M. Proietti and A. Pettorossi. Unfolding – definition – folding, in this order for avoiding unnecessary variables in logic programs. In *Proceedings PLILP'91*, pages 347–358. Springer-Verlag, LNCS 528, 1991.

34. M. Proietti and A. Pettorossi. The loop absorption and the generalization strategies for the development of logic programs and partial deduction. *Journal of Logic Programming*, 16:123–161, 1993.

35. M. Proietti and A. Pettorossi. Completeness of some transformation strategies for avoiding unnecessary logical variables. In P. Van Hentenryck, editor, *Proceedings ICLP'94*, pages 714–729, Italy, June 1994. MIT Press.

36. M.H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In J. Lloyd, editor, *Proceedings ILPS'95*, pages 465–479, Portland, Oregon, December 1995, MIT Press.

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

38. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury: An efficient purely declarative logic programming language. *The Journal of Logic Programming*, 1996. To Appear.

39. H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In S-Å. Tärnlund, editor, *Proceedings ICLP'84*, pages 127–138, Uppsala, July 1984.

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

41. V.F. Turchin. The algorithm of generalization in the supercompiler. In D. Bjørner, A.P. Ershov and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 531–549. North-Holland, 1988.

42. P.L. Wadler. Deforestation: Transforming programs to eliminate intermediate trees. *Theoretical Computer Science*, 73:231–248, 1990.

A Benchmark Programs

The benchmark programs were carefully selected and/or designed in such a way that they cover a wide range of different application areas, including: pattern matching, databases, expert systems, meta-interpreters (non-ground vanilla, mixed, ground), and more involved particular ones: a model-elimination theorem prover, the missionaries-cannibals problem, a meta-interpreter for a simple imperative language. The benchmarks marked with a star (*) can be fully unfolded. Full descriptions can be found in [20].

Benchmark	Description
advisor*	A very simple expert system - benchmark by Thomas Horváth [17].
applast	The append-last program.
contains.kmp	A benchmark based on the "contains" Lam & Kusalik benchmark [18], but with improved run-time queries.
depth.lam*	A Lam & Kusalik benchmark [18].
doubleapp	The double append example. Tests whether deforestation can be done.
ex_depth	A variation of *depth.lam* with a more sophisticated object program.
flip	A simple deforestation example from Wadler [42].
grammar.lam	A Lam & Kusalik benchmark [18].
groundunify.complex	A ground unification algorithm calculating explicit substitutions [10].
groundunify.simple*	A ground unification algorithm calculating explicit substitutions.
imperative.power	A solver for a simple imperative language. Specialise a power sub-program for a known power and base but unknown environment.
liftsolve.app	The lifting meta-interpreter for the ground representation with append as object program.
liftsolve.db1*	The lifting meta-interpreter [14] with a simple, fully unfoldable object program.
liftsolve.db2	The lifting meta-interpreter with a partially specified object program.
liftsolve.lmkng	Testing part of lifting meta-interpreter (generates an ∞ number of chtrees).
map.reduce	Specialising the higher-order map/3 (using call and =..) for the higher-order reduce/4 in turn applied to add/3.
map.rev	Specialising the higher-order map for the reverse program.
match-append	A very naive matcher, written using 2 appends. Same queries as match.kmp.
match.kmp	Try to obtain a KMP matcher. A benchmark based on the "match" Lam & Kusalik benchmark [18] but with improved run-time queries.
maxlength	Tests whether tupling can be done.
memo-solve	A variation of ex_depth with a simple loop prevention mechanism based on keeping a call stack.
missionaries	A program for the missionaries and cannibals problem.
model_elim.app	Specialise the Poole & Goebel [31] model elimination prover (also used by de Waal & Gallagher [11]) for the append program.
regexp.r1	A naive regular expression matcher. Regular expression: (a+b)*aab.
regexp.r2	Same program as regexp.r1 for ((a+b)(c+d)(e+f)(g+h))*.
regexp.r3	Same program as regexp.r1 for ((a+b)(a+b)(a+b)(a+b)(a+b)(a+b))*.
relative.lam*	A Lam & Kusalik benchmark [18].
remove	A sophisticated deforestation example.
remove2	An even more sophisticated deforestation example. Adapted from Turchin [41].
rev_acc_type	A simple benchmark generating an ∞ number of different characteristic trees.
rev_acc_type.inffail	A simple benchmark with infinite determinate failure at pe time.
rotateprune	A more sophisticated deforestation example [33].
ssuply.lam*	A Lam & Kusalik benchmark [18].
transpose.lam*	A Lam & Kusalik benchmark [18].
upto.sum1	Calculates the sum of squares for 1 up to n. Adapted from Wadler [42].
upto.sum2	Calculates the square of integers in nodes of a tree and sums these up. Adapted from Wadler [42].

Table 3. Description of the benchmark programs

17

B Benchmark Results

We benchmarked time and size of compiled code under Prolog by BIM 4.0.12. The timings were not obtained via a loop with an overhead but via special prolog files (generated automatically by our partial deduction system). These files call the original and specialised programs directly (i.e. without overhead) at least 100 times for the respective run-time queries. The timings were obtained via the *time*/2 predicate of Prolog by BIM 4.0.12 on a Sparc Classic under Solaris. The compiled code size was obtained via *statistics*/4 and is expressed in units, where 1 unit = 4.08 bytes (in the current implementation of Prolog by BIM).

All timings were for renamed queries, except for the original and for SP (which does not rename the top-level query — this puts SP at a slight disadvantage of about 10% in average). Note that Paddy systematically included the original program and the specialised part could only be called in a renamed style. We removed the original program whenever possible and added 1 clause which allows calling the specialised program in an unrenamed style as well (just like Mixtus and Ecce) to avoid distortion in the code size (and speedup) figures.

Runtimes (RT) are given relative to the runtimes of the original programs. In computing averages and totals, time and size of the original program were taken in case of non-termination or an error occurring during transformation. The total speedups are obtained by the formula

$$\frac{n}{\sum_{i=1}^{n} \frac{spec_i}{orig_i}}$$

where $n = 36$ is the number of benchmarks and $spec_i$ and $orig_i$ are the absolute execution times of the specialised and original programs respectively. The weighted total speedups are obtained by using the code size $size_i$ of the original program as a weight for computing the average:

$$\frac{\sum_{i=1}^{n} size_i}{\sum_{i=1}^{n} size_i \frac{spec_i}{orig_i}}$$

TT is the transformation time in seconds.

Timing in (BIM) Prolog, especially on Sparc machines, can sometimes be problematic. For instance, for maxlength, deforestation does not seem to pay off. However, with reordering of clauses we go from a relative time of 1.4 (i.e. a slowdown) to a relative time of 0.9 (i.e. a speedup)! On Sicstus Prolog 3, we even get a 20 % speedup for this example (without reordering)! The problem is probably due to the caching behaviour of the Sparc processor.

The following versions of the existing systems have been used: version 0.3.3 of MIXTUS , the version of PADDY delivered with ECLIPSE 3.5.1 and a version of SP dating from September 25th, 1995. We briefly explain the use of ∞ in the tables:

- ∞, SP: this means real non-termination
- ∞, MIXTUS: heap overflow after 20 minutes
- ∞, PADDY: thorough system crash after 2 minutes

18

It seems that the latest version 0.3.6 of MIXTUS does terminate for the missionaries example, but we did not yet have time to redo the experiments. PADDY and SP did not terminate for one other example (memo-solve and imperative.power respectively) when we accidentally used *not* instead of \+ (*not* is not defined in SICStus Prolog; PADDY and SP follow this convention). After changing to \+, both systems terminated.

$> 12h$ means that the specialisation was interrupted after 12 hours (though, theoretically, it should have terminated by itself when granted sufficient time to do so). **bi err** means that an error occured while running the program due to a call of a built-in where the arguments were not sufficiently instantiated.

Finally, a brief remark on the match-append benchmark. The bad figures of most systems seem to be due to a bad choice of the filtering, further work will be needed the avoid this kind of effect. Also, none of the presented methods was able to deforest this particular example. However, if we run for instance Csc-hh-t twice on match-append we get the desired deforestation and a much improved performance (relative time of 0.03 !). It should be possible to get this effect directly by using a more refined control.

19

Benchmark	Cdc-hh-t (DynamicContig-hh-det)			Csc-hh-t (StaticContig-hh-det)			Csc-th-t (StaticContig-th-det)		
	RT	Size	TT	RT	Size	TT	RT	Size	TT
advisor	0.47	412	0.90	0.47	412	0.86	0.47	412	0.87
applast	0.36	202	0.92	0.36	202	0.80	0.36	202	0.67
contains.kmp	0.11	1039	5.61	0.11	1039	5.41	0.11	1039	5.44
depth.lam	0.15	1837	4.11	0.15	1837	4.01	0.15	1837	3.82
doubleapp	0.80	362	0.85	0.80	362	0.88	0.80	362	0.84
ex_depth	0.26	508	3.30	0.29	407	1.62	0.29	407	1.60
flip	1.33	686	1.41	1.33	686	1.25	1.33	686	1.02
grammar.lam	0.16	309	1.94	0.16	309	1.84	0.16	309	1.82
groundunify.complex	0.40	6247	118.69	0.47	6277	19.47	0.47	6277	19.08
groundunify.simple	0.25	368	0.78	0.25	368	0.80	0.25	368	0.75
imperative.power	0.40	36067	906.60	0.40	3132	71.37	0.40	3293	42.85
liftsolve.app	0.05	1179	5.75	0.05	1179	5.98	0.05	1179	5.74
liftsolve.db1	0.01	1280	22.39	0.01	1280	14.23	0.01	1280	13.33
liftsolve.db2	0.16	17472	2599.03	0.21	21071	1594.90	0.17	5929	198.19
liftsolve.lmkng	1.02	1591	3.09	1.02	1591	2.66	1.02	1591	2.63
map.reduce	0.07	507	0.78	0.07	507	0.85	0.07	507	0.80
map.rev	0.11	427	0.83	0.11	427	0.82	0.11	427	0.80
match-append	1.21	406	1.29	1.21	406	1.14	1.21	406	1.17
match.kmp	0.73	639	1.16	0.73	639	1.15	0.73	639	1.15
maxlength	1.40	620	1.22	1.40	620	1.14	1.40	620	1.17
memo-solve	0.81	1095	5.88	0.81	1095	2.53	0.81	1095	4.54
missionaries	0.69	2960	7.93	0.69	2960	7.59	0.69	2960	7.13
model_elim.app	0.12	451	2.65	0.12	451	2.66	0.12	451	2.58
regexp.r1	0.39	557	1.76	0.39	557	1.36	0.39	557	1.41
regexp.r2	0.41	833	3.57	0.53	692	1.52	0.53	692	1.55
regexp.r3	0.31	1197	6.85	0.44	873	1.82	0.44	873	1.87
relative.lam	0.07	1011	5.80	0.07	1011	5.39	0.07	1011	5.32
remove	0.62	1774	5.34	0.62	1774	4.89	0.62	1774	4.92
remove2	0.87	1056	3.42	0.92	831	2.08	0.92	831	2.13
rev_acc_type	1.00	242	1.01	1.00	242	0.91	1.00	242	0.96
rev_acc_type.inffail	0.63	864	3.21	0.63	864	3.01	0.63	864	3.09
rotateprune	0.71	1165	3.08	0.71	1165	2.80	0.71	1165	2.81
ssuply.lam	0.06	262	1.31	0.06	262	1.17	0.06	262	1.19
transpose.lam	0.17	2312	2.87	0.17	2312	2.45	0.17	2312	2.53
upto.sum1	1.20	848	4.00	1.20	848	3.64	0.88	734	3.03
upto.sum2	1.12	623	1.48	1.12	623	1.46	1.12	623	1.48
Average	0.52	2484	103.91	0.53	1648	49.35	0.52	1228	9.73
Total	18.66	89408	3740.8	19.10	59311	1776.5	18.73	44216	350.3
Total Speedup	**1.93**			**1.89**			**1.92**		
Weighted Speedup	**2.44**			**2.38**			**2.44**		

Table 4. Ecce Determinate Conjunctive Partial Deduction (A)

20

Benchmark	Csc-hn-t (StaticContig-hn-det)			Csc-tn-t (StaticContig-tn-det)			Cdc-th-t (DyanmicContig-th-det)		
	RT	Size	TT	RT	Size	TT	RT	Size	TT
advisor	0.47	412	0.88	0.47	412	1.20	0.47	412	0.82
applast	0.36	202	0.64	0.36	202	0.73	0.36	202	0.86
contains.kmp	0.11	1039	5.19	0.63	862	1.16	0.11	1039	5.22
depth.lam	0.15	1837	3.95	0.15	1837	4.18	0.15	1837	3.53
doubleapp	0.80	362	0.86	0.80	362	1.13	0.80	362	0.86
ex_depth	0.29	407	1.60	0.29	407	1.75	0.27	508	3.08
flip	1.33	686	1.07	1.33	686	1.14	1.33	686	1.41
grammar.lam	0.16	309	1.77	0.16	309	1.98	0.16	309	1.76
groundunify.complex	0.40	4869	15.03	0.47	5095	18.67	0.40	6247	81.35
groundunify.simple	0.25	368	0.76	0.25	368	0.94	0.25	368	0.73
imperative.power	0.37	2881	49.33	0.37	2881	32.88	0.40	37501	1039.48
liftsolve.app	0.05	1179	5.50	0.05	1179	5.65	0.05	1179	5.43
liftsolve.db1	0.01	1280	13.60	0.01	1280	13.56	0.01	1280	20.39
liftsolve.db2	0.17	10146	1974.50	0.33	3173	19.84	0.17	11152	628.47
liftsolve.lmkng	1.09	1416	1.82	1.07	1416	2.09	1.02	1591	2.89
map.reduce	0.07	507	0.83	0.07	507	1.09	0.07	507	0.77
map.rev	0.11	427	0.79	0.11	427	1.04	0.11	427	0.80
match-append	1.21	406	0.91	1.21	406	1.06	1.21	406	1.18
match.kmp	0.73	639	1.11	0.73	613	1.69	0.73	639	1.22
maxlength	1.40	620	1.20	1.40	620	1.31	1.40	620	1.13
memo-solve	0.81	1095	4.28	1.38	1709	6.73	0.81	1095	10.32
missionaries	0.69	2960	7.06	0.71	3083	6.03	0.69	2960	7.86
model_elim.app	0.12	451	2.63	0.12	451	2.75	0.12	451	2.60
regexp.r1	0.39	557	1.38	0.39	557	1.84	0.39	557	1.76
regexp.r2	0.53	692	1.55	0.53	692	1.66	0.43	833	3.55
regexp.r3	0.44	873	1.81	0.44	873	2.00	0.30	1197	6.01
relative.lam	0.07	1011	5.34	0.45	1252	6.78	0.07	1011	5.74
remove	0.65	1191	2.42	0.65	1191	2.19	0.62	1774	5.39
remove2	0.92	831	2.04	0.92	831	1.89	0.87	1056	3.45
rev_acc_type	1.00	242	0.67	1.00	242	0.84	1.00	242	1.04
rev_acc_type.inffail	0.63	598	0.87	0.63	598	0.98	0.63	864	3.24
rotateprune	0.71	1165	2.79	0.71	1165	2.55	0.71	1165	3.10
ssuply.lam	0.06	262	1.15	0.06	262	1.32	0.06	262	1.32
transpose.lam	0.17	2312	2.46	0.17	2312	2.62	0.17	2312	2.51
upto.sum1	1.20	848	3.77	0.88	734	2.83	0.88	734	3.40
upto.sum2	1.12	623	1.45	1.12	623	1.39	1.12	623	1.49
Average	0.53	1270	58.97	0.57	1100	4.37	0.51	2345	51.78
Total	19.04	45703	2123.01	20.40	39617	157.49	18.35	84408	1864.16
Total Speedup	**1.89**			**1.76**			**1.96**		
Weighted Speedup	**2.40**			**2.18**			**2.49**		

Table 5. Ecce Determinate Conjunctive Partial Deduction (B)

21

	Csc-th-li (StaticContig-th-lidx) (Non-det. unfolding)			Cdm-hh-t (DynamicMcs-hh-det) (Cautious Deforestation)			Csm-hh-h (StaticMcs-hh-hrs) (Aggressive Deforestation)		
Benchmark	RT	Size	TT	RT	Size	TT	RT	Size	TT
advisor	0.32	809	0.85	0.47	412	0.80	0.46	647	1.00
applast	0.34	145	0.67	0.36	202	0.97	0.36	145	0.85
contains.kmp	0.10	1227	15.73	0.11	1039	5.16	0.10	814	5.77
depth.lam	0.15	1848	12.65	0.15	1837	3.99	0.15	1848	5.91
doubleapp	0.82	277	0.98	0.80	362	0.84	0.82	277	0.83
ex_depth	0.27	659	6.31	0.26	508	3.15	0.34	1240	6.18
flip	0.95	493	1.26	0.75	441	1.11	0.69	267	0.81
grammar.lam	0.14	218	2.15	0.16	309	1.74	0.16	309	2.21
groundunify.complex	0.47	19640	117.12	0.40	6247	137.68	0.47	8113	77.21
groundunify.simple	0.25	368	0.76	0.25	368	0.75	0.25	399	1.27
imperative.power	0.69	3605	155.55	0.37	103855	4548.51	bi err	85460	1617.93
liftsolve.app	0.05	1179	6.04	0.05	1179	5.44	0.06	1210	6.26
liftsolve.db1	0.02	1326	19.94	0.01	1280	20.85	0.02	1311	18.10
liftsolve.db2	-	-	$> 12h$	0.17	17206	1813.49	-	-	$> 12h$
liftsolve.lmkng	1.00	1591	4.22	1.02	1591	2.85	1.24	1951	8.89
map.reduce	0.08	348	0.78	0.07	507	0.85	0.08	348	0.84
map.rev	0.13	285	0.79	0.11	427	0.88	0.13	285	0.75
match-append	1.36	362	1.02	1.21	406	1.20	1.36	362	0.98
match.kmp	0.65	543	1.65	0.73	639	1.18	0.65	543	0.94
maxlength	1.30	620	1.22	1.40	620	1.18	1.10	314	1.19
memo-solve	0.95	1015	3.69	1.12	1294	22.69	1.50	3777	34.84
missionaries	0.54	15652	348.68	-	-	$> 12h$	21.17	43268	2537.39
model_elim.app	0.13	444	3.11	0.12	451	2.77	0.12	451	3.20
regexp.r1	0.20	457	1.04	0.39	557	1.91	0.20	457	1.21
regexp.r2	0.41	831	4.98	0.41	833	3.65	0.57	1954	6.43
regexp.r3	0.31	1041	14.70	0.31	1197	6.84	1.89	9124	31.07
relative.lam	0.00	261	6.19	0.07	1011	5.84	0.01	954	7.31
remove	0.87	1369	7.31	0.62	1774	5.51	6.40	4116	7.69
remove2	0.93	862	3.51	0.87	1056	3.65	0.94	862	2.34
rev_acc_type	1.00	242	1.21	1.00	242	1.08	1.00	242	1.73
rev_acc_type.inffail	0.66	700	2.24	0.63	864	3.26	0.61	786	1.94
rotateprune	0.80	1470	4.62	0.71	1165	4.01	0.17	691	2.33
ssuply.lam	0.06	262	1.41	0.06	262	1.36	0.06	262	1.69
transpose.lam	0.18	2312	2.99	0.17	2312	2.77	0.19	2436	4.34
upto.sum1	0.88	734	2.81	bi err	448	3.43	bi err	479	3.79
upto.sum2	1.00	654	1.48	0.65	394	1.50	0.58	242	1.11
Average	0.53	1824	21.70	0.50	4380	189.23	1.30	5027	125.90
Total	19.03	63849	759.66	18.01	153295	6622.9	46.84	175944	4406.33
Total Speedup	**1.89**			**2.00**			**0.77**		
Weighted Speedup	**2.38**			**2.39**			**0.52**		

Table 6. Ecce Non-Determinate Conjunctive Partial Deduction

Benchmark	S-hh-t (StdPD-hh-det)			S-hh-li (StdPD-hh-lidx)			SE-hh-x (StdEcoPD-hh-mixtus)		
	RT	Size	TT	RT	Size	TT	RT	Size	TT
advisor	0.47	412	0.87	0.31	809	0.85	0.31	809	0.78
applast	1.05	343	0.71	1.48	314	0.75	1.48	314	0.70
contains.kmp	0.85	1290	2.69	0.55	1294	5.97	0.09	685	4.48
depth.lam	0.94	1955	1.47	0.62	1853	3.76	0.02	2085	1.91
doubleapp	0.95	277	0.65	0.95	216	0.58	0.95	216	0.53
ex_depth	0.76	1614	2.54	0.44	1649	4.26	0.32	350	1.58
flip	1.05	476	0.77	1.03	313	0.65	1.03	313	0.53
grammar.lam	0.16	309	1.91	0.14	218	2.43	0.14	218	1.90
groundunify.complex	0.40	5753	13.47	0.47	8356	50.63	0.53	4800	0.75
groundunify.simple	0.25	368	0.78	0.25	368	0.77	0.25	368	22.03
imperative.power	0.42	2435	75.10	0.58	2254	62.97	0.54	1578	27.42
liftsolve.app	0.05	1179	6.05	0.06	1179	6.40	0.06	1179	6.57
liftsolve.db1	0.01	1280	13.27	0.02	1326	20.82	0.02	1326	7.33
liftsolve.db2	0.18	3574	16.86	0.76	3751	242.86	0.61	4786	34.25
liftsolve.lmkng	1.07	1730	1.80	1.07	1730	2.12	1.02	2385	2.75
map.reduce	0.07	507	0.91	0.08	348	0.82	0.08	348	0.86
map.rev	0.11	427	0.83	0.13	285	0.88	0.11	427	0.89
match-append	1.21	406	0.64	1.36	362	0.75	1.36	362	0.68
match.kmp	0.73	639	1.16	0.65	543	1.77	0.70	669	1.23
maxlength	1.20	715	1.07	1.10	715	1.16	1.10	421	0.95
memo-solve	1.17	2318	4.74	1.20	2238	4.96	1.09	2308	4.31
missionaries	0.81	2294	5.11	0.66	13168	430.99	0.72	2226	9.21
model_elim.app	0.63	2100	2.82	0.13	444	3.18	0.13	532	3.56
regexp.r1	0.50	594	1.28	0.20	457	1.03	0.29	435	0.98
regexp.r2	0.57	629	1.28	0.61	737	4.67	0.51	1159	4.87
regexp.r3	0.50	828	1.74	0.38	961	14.00	0.42	1684	14.92
relative.lam	0.82	1074	1.89	0.00	261	5.88	0.00	261	4.06
remove	0.71	955	1.46	0.68	659	1.02	0.68	659	0.90
remove2	0.74	508	1.15	0.75	453	1.30	0.80	440	1.00
rev_acc_type	1.00	242	0.70	1.00	242	0.92	1.00	242	0.83
rev_acc_type.inffail	0.63	864	1.48	0.80	850	1.25	0.60	527	0.80
rotateprune	0.71	1165	1.77	1.02	779	1.10	1.02	779	0.88
ssuply.lam	0.06	262	1.15	0.06	262	1.51	0.06	262	1.18
transpose.lam	0.17	2312	2.49	0.17	2312	2.99	0.17	2312	1.98
upto.sum1	1.06	581	2.18	1.07	581	1.80	1.20	664	3.11
upto.sum2	1.10	623	1.50	1.05	485	1.38	1.05	485	0.94
Average	0.64	1196	4.90	0.61	1466	24.70	0.57	1073	4.77
Total	23.12	43038	176.29	21.86	52772	889.18	20.47	38614	171.65
Total Speedup	**1.56**			**1.65**			**1.76**		
Weighted Speedup	**1.86**			**2.09**			**2.24**		

Table 7. Ecce Standard Partial Deduction Methods

Benchmark	Mixtus RT	Size	TT	PADDY RT	Size	TT	SP RT	Size	TT
advisor	0.31	809	0.85	0.31	809	0.10	0.40	463	0.29
applast	1.27	309	0.28	1.30	309	0.08	0.84	255	0.15
contains.kmp	0.16	533	2.48	0.11	651	0.55	0.75	985	1.13
depth.lam	0.04	1881	4.15	0.02	2085	0.32	0.53	928	0.99
doubleapp	1.00	295	0.30	0.98	191	0.08	1.02	160	0.11
ex_depth	0.40	643	2.40	0.29	1872	0.53	0.27	786	1.35
flip	1.03	495	0.37	1.02	290	0.12	1.02	259	0.13
grammar.lam	0.17	841	2.73	0.43	636	0.22	0.15	280	0.71
groundunify.complex	0.67	5227	11.68	0.60	4420	1.53	0.73	4050	2.46
groundunify.simple	0.25	368	0.45	0.25	368	0.13	0.61	407	0.20
imperative.power	0.56	2842	5.35	0.58	3161	2.18	1.16	1706	6.97
liftsolve.app	0.06	1179	4.78	0.06	1454	0.80	0.23	1577	2.46
liftsolve.db1	0.01	1280	5.36	0.02	1280	1.20	0.82	4022	3.95
liftsolve.db2	0.31	8149	58.19	0.32	4543	1.60	0.82	3586	3.71
liftsolve.lmkng	1.16	2169	4.89	0.98	1967	0.32	1.16	1106	0.37
map.reduce	0.68	897	0.17	0.08	498	0.20	0.09	437	0.23
map.rev	0.11	897	0.16	0.26	2026	0.37	0.13	351	0.20
match-append	0.47	389	0.27	0.98	422	0.12	0.99	265	0.18
match.kmp	1.55	467	4.89	0.69	675	0.28	1.08	527	0.49
maxlength	1.20	594	0.72	0.90	398	0.17	0.90	367	0.31
memo-solve	0.60	1493	12.72	1.48	3716	1.70	1.15	1688	3.65
missionaries	-	-	∞	-	-	∞	0.73	16864	82.59
model_elim.app	0.13	624	5.73	0.10	931	0.90	-	-	∞
regexp.r1	0.20	457	0.73	0.29	417	0.13	0.54	466	0.37
regexp.r2	0.82	1916	2.85	0.67	3605	0.63	1.08	1233	0.67
regexp.r3	0.60	2393	4.49	1.26	10399	1.35	1.03	1646	1.20
relative.lam	0.01	517	7.76	0.00	517	0.42	0.69	917	0.35
remove	0.81	715	0.49	0.71	437	0.12	0.75	561	0.29
remove2	1.01	715	0.84	0.84	756	0.12	0.82	386	0.25
rev_acc_type	1.00	497	0.99	0.99	974	0.33	-	-	∞
rev_acc_type.inffail	0.97	276	0.77	0.94	480	0.28	-	-	∞
rotateprune	1.02	756	0.49	1.01	571	0.12	1.00	725	0.31
ssuply.lam	0.06	262	0.93	0.08	262	0.08	0.06	231	0.52
transpose.lam	0.18	1302	3.89	0.18	1302	0.43	0.26	1267	0.52
upto.sum1	0.96	556	1.80	1.08	734	0.30	1.05	467	0.48
upto.sum2	1.06	462	0.44	1.06	462	0.13	1.01	431	0.21
Average	0.61	1234	4.44	0.61	1532	0.51	0.75	1497	3.57
Total	21.83	43205	155.37	21.87	53618	17.95	26.86	49399	117.80
Total Speedup	1.65			1.65			1.34		
Weighted Speedup	2.11			2.00			1.54		

Table 8. Existing systems