Fully Automatic Binding-Time Analysis for Prolog*

Stephen-John Craig[1], John P. Gallagher[2], Michael Leuschel[1], and Kim S. Henriksen[2]

[1] Department of Electronics and Computer Science, University of Southampton, Highfield, Southampton, SO17 1BJ, UK
{sjc02r, mal}@ecs.soton.ac.uk
[2] Department of Computer Science,University of Roskilde,**
P.O. Box 260, D-4000 Roskilde, Denmark
{jpg, kimsh}@ruc.dk

Abstract. Offline partial evaluation techniques rely on an annotated version of the source program to control the specialisation process. These annotations guide the specialisation and ensure the termination of the partial evaluation. We present an algorithm for generating these annotations automatically. The algorithm uses state-of-the-art termination analysis techniques, combined with a new type-based abstract interpretation for propagating the binding types. This algorithm has been implemented as part of the LOGEN partial evaluation system, along with a graphical annotation visualiser and editor, and we report on the performance of the algorithm for a series of benchmarks.

1 Introduction

The offline approach to specialisation has proven to be very successful for functional and imperative programming, and more recently for logic programming.

Most offline approaches perform a *binding-time analysis* (BTA) prior to the specialisation phase. Once this has been performed, the specialisation process itself can be done very efficiently [20] and with a predictable outcome.

Compared to online specialisation, offline specialisation is in principle less powerful (as control decisions are taken by the BTA *before* the actual static input is available), but much more efficient (once the BTA has been performed). This makes offline specialisation very useful for compiling interpreters [19], a key application of partial evaluation. However, up until now, no automatic BTA for logic programs has been fully implemented (though there are some partial implementations, discussed in Section 7), requiring users to manually annotate

* Work supported in part by European Framework 5 Project ASAP (IST-2001-38059).
** Roskilde authors supported in part by the IT-University of Copenhagen.

S. Etalle(Ed.): LOPSTR 2004, LNCS 3573, pp. 53–68, 2005.

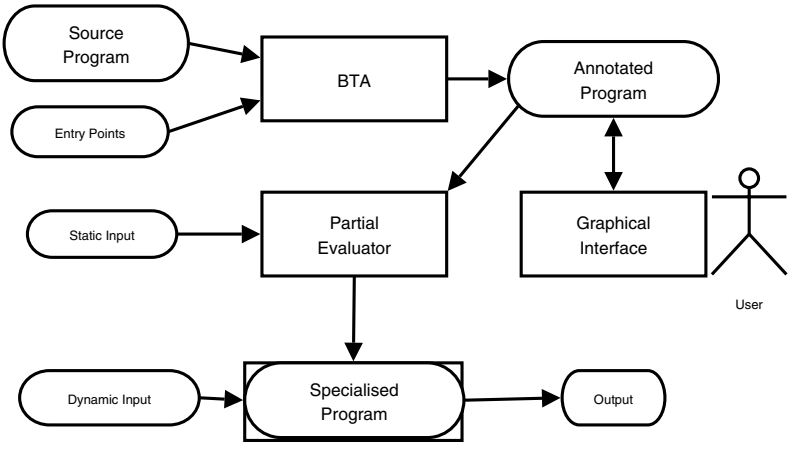

Fig. 1. The role of the BTA for offline specialisation using LOGEN

the program. This is an error-prone process, and requires considerable expertise. Hence, to make offline specialisation accessible to a wider audience, a fully automatic BTA is essential.

In essence, a *binding-time analysis* does the following: given a program and a description of the input available for specialisation, it approximates all values within the program and generates annotations that steer the specialisation process. The partial evaluator (or the compiler generator generating the specialised partial evaluator) then uses the generated annotated program to guide the specialisation process. This process is illustrated in Fig. 1. The figure also shows our new graphical editor which allows a user to inspect the annotations and fine tune them if necessary.

To guide our partial evaluator the binding-time analysis must provide *binding types* and *clause annotations*, which will now be described.

Binding Types

Each argument of a predicate in an annotated program is given a *binding type* by means of *filter declarations.*A binding type indicates something about the structure of an argument at specialisation time. The basic binding types are usually known as *static* and *dynamic* defined as follows.

- **static:** The argument is definitely known at specialisation time;
- **dynamic:** The argument is possibly unknown at specialisation time.

We will see in Section 3 that more precise binding types can be defined by means of regular type declarations, and combined with basic binding types. For example, an interpreter may use an environment that is a partially static data structure at partial evaluation time. To model the environment, e.g., as a list of static names mapped to dynamic variables we would use the following definition:

```
:- type binding = static / dynamic.
:- type list_env = [] ;  [binding | list_env].
```

Through the filter declarations we associate binding types with arguments of particular predicates, as in the following example (taken from the `inter_binding` benchmark to be discussed in Section 6):

```
:- filter int(static, (type list_env), dynamic).
```

The filter declarations influence *global control*, since *dynamic* parts of arguments are generalised away (that is, replaced by fresh variables) and the known, *static* parts are left unchanged. They also influence whether arguments are "filtered out" in the specialised program. Indeed, static parts are already known at specialisation time and hence do not have to be passed around at runtime.

Clause Annotations
Clause annotations indicate how each call in the program should be treated during specialisation. Essentially, these annotations determine whether a call in the body of a clause is performed at specialisation time or at run time. Clause annotations influence the *local control* [22]. For the LOGEN system [20] the main annotations are the following.

- **Unfold:** The call is unfolded under the control of the partial evaluator. The call is replaced with the predicate body, performing all the needed substitutions
- **Memo:** The call is not unfolded, instead the call is generalised using the filter declaration and specialised independently
- **Call:** The call is fully executed without further intervention
- **Rescall:** The call is left unmodified in the residual code

2 Algorithm Overview

Implementing a fully automatic BTA is a challenging task for several reasons. First, the binding type information about the static and dynamic parts of arguments has to be propagated all throughout the program. Second, on has to decide how to treat each body call in the program. This has to be guided by termination issues (avoiding infinite unfolding) but also safety issues (avoiding calling built-ins that are not sufficiently instantiated). Furthermore, the decisions made about how to treat body calls in turn affect the propagation of the binding types, which in turn affect how body calls are to be treated. In summary, we need

- a precise way to propagate binding types, allowing for new types and partially static data,
- a way to detect whether the current annotations ensure safety and termination at specialisation time,
- and an overall algorithm to link the above two together.

Also, in case the current annotations do not ensure termination we need a way to identify the body calls that are causing the (potential) non-termination in order to update the annotations. For this we had to implement our own termination analyzer, based on the binary clause semantics [9]. To achieve a precise propagation of binding types we have used a new analysis framework [13] based on regular types and type determinization.

We now outline the main steps of our overall BTA algorithm depicted in Fig. 2. The input to the algorithm consists of a program, a set of binding types, and a filter declaration giving binding types to the entry query (the query with respect to which the program is to be partially evaluated). The core of the algorithm is a loop which propagates the binding types from the entry query with respect to the current clause annotations (step 1), generates the abstract binary program (steps 2 and 3) and checks for termination conditions (step 4).

If a call is found to be unsafe at step 4 (e.g. might not terminate) the annotations are modified accordingly. Initially, all calls are annotated as *unfold* (or *call* for built-ins), with the exception of imported predicates which are annotated as *rescall* (step 0). Annotations can be changed to *memo* or *rescall*, until termination is established. Termination of the main loop is ensured since there is initially only a finite number of *unfold* or *call* annotations, and each iteration of the loop eliminates one or more *unfold* or *call* annotation.

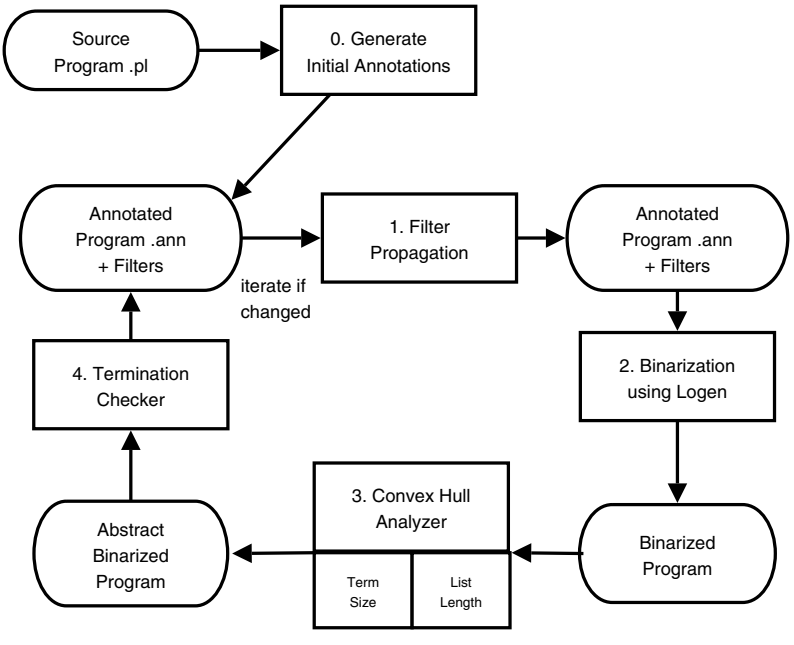

Fig. 2. Overview of the BTA algorithm

The decision on how to annotate calls to built-in predicates cannot be handled by the termination checker, but is guided by a definition of the allowed calling patterns, with respect to the given set of binding types. For instance, considering simple binding types *static* and *dynamic*, the call `X > Y` can be executed only when both `X` and `Y` are static, whereas the call `X is Y` can be executed where `Y` is static but `X` is dynamic (either known or unknown). Some built-ins have more than one allowed calling pattern; for example `functor(T,F,N)` can be executed if either `T` is static or both `F` and `N` are static. Whenever the binding types for a call to a built-in predicate do not match one of the allowed calling patterns, the call is marked *rescall*. Thus if no calling patterns are supplied for some built-in, then all calls to that built-in will be annotated *rescall*.

3 Binding Type Propagation

The basis of the BTA is a classification of arguments using abstract values. In this section we explain how to obtain such a classification for a given program and initial goal. Our method is completely independent of the actual binding types, apart from requiring that they should include the type *dynamic*. Usually *static* and *nonvar* are also included. A *binding-time division* is a set of filter declarations of the form $p(t_1, \ldots, t_n)$, where p/n is a program predicate and $t_1, \ldots, t_n$ are binding types. For the purpose of explanation we consider here only *monovariant* binding-time divisions, namely those in which there is not more than one filter declaration for each predicate. However, the algorithm has been extended to *polyvariant* binding-time divisions, which allow several filter declarations for each predicate.

A binding-time division defines the binding types occurring in each predicate call in an execution of the program for a given initial goal. This information in turn is used when determining which calls to unfold and which to keep in the residual programs. A binding-time division should be *safe* in the sense that every possible concrete call is described by some filter declaration in it.

The use of *static-dynamic* binding types was introduced for functional programs, and has been used in BTAs for logic programs [23]. However, a simple classification of arguments into "fully known" or "totally unknown" is often unsatisfactory in logic programs, where partially unknown terms occur frequently at runtime, and would prevent specialisation of many "natural" logic programs such as the vanilla meta-interpreter [15, 21] or most of the benchmarks from the DPPD library [18].

We outline a method of describing more expressive binding types and propagating them. The analysis framework is described elsewhere [13]. In this framework, modes (or "binding times") such as *static* and *dynamic* can be freely mixed with binding *types* such as lists.

Regular Binding Types

A regular type t is defined by a rule $t = f_1(t_{1,1}, \ldots, t_{1,m_1}); \ldots; f_n(t_{n,1}, \ldots, t_{n,m_n})$ where $f_1, \ldots, f_n$ are function symbols (possibly not distinct) and for all $1 \leq i \leq n$ and $1 \leq j \leq m_i$, m_i is the arity of f_i, and $t_{i,j}$ are regular types. The

interpretation of such rules is well understood in the framework of regular tree grammars or finite tree automata [10].

Instantiation modes including *static*, *dynamic* and *nonvar* can be coded as regular types, for a fixed signature. For example, if we assume that the signature is $\{[], [.|.], s, 0, v\}$ with the usual arities, then the definitions of the types *ground term* (*static*), *non-variables* (*nonvar*) and *any term* (*dynamic*) are as follows.

$$static = 0; []; [static|static]; s(static)$$
$$nonvar = 0; []; [dynamic|dynamic]; s(dynamic)$$
$$dynamic = 0; []; [dynamic|dynamic]; s(dynamic); v$$

The constant v is a distinguished constant not occurring in programs or goals. Note that v is not included in the types *static* and *nonvar*. Therefore any term of type *dynamic* is possibly a variable.

In addition to modes, regular types can describe common data structures. The set of all lists, for instance, is given as $list = []; [dynamic|list]$. We can describe the set of lists of list by the type $listlist = []; [list|listlist]$. Program-specific types such as the type of environments are also regular types.

$$binding = static/dynamic$$
$$list\_env = []; [binding|list\_env]$$

Type Determinization

We take a given set of regular types, and transform them into a set of *disjoint* regular types. This process is called *determinization* and is a standard operation on finite tree automata [10]. A set of disjoint types is represented by a set of type rules of the form $t = f_1(t_{1,1}, \ldots, t_{1,m_1}); \ldots; f_n(t_{n,1}, \ldots, t_{n,m_n})$ as before, but with the added condition that there are no two rules having an occurrence of the same term $f_i(t_{i,1}, \ldots, t_{i,m_i})$. Such a set of rules corresponds to a bottom-up deterministic finite tree automata [10]. The inclusion of the type *dynamic* ensures that the set of rules is *complete*, that is, that every term is a member of exactly one of the disjoint types.

For example, given the types *dynamic*, *static*, *nonvar* and *list* as shown above, determinization yields definitions of the disjoint sets of terms, which are (1) non-ground, non-variable non-lists, (2) non-ground lists, (3) ground lists, (4) variables and (5) ground non-lists. The rules defining these disjoint types are typically hard to read and would be difficult to write directly. For example, naming the above 5 types $q_1, \ldots, q_5$ respectively, the type rule for non-ground lists is $q_2 = [q_1|q_2]; [q_2|q_2]; [q_3|q_2]; [q_4|q_2]; [q_5|q_2]; [q_2|q_3]; [q_1|q_3]; [q_4|q_3]$. A more compact representation is actually used [13].

Types are abstractions of terms, and can be used to construct a domain for abstract interpretation [4, 7, 8, 16]. The advantage of determinized types is that we can propagate them more precisely than non-disjoint types. Overlapping types tend to lose precision. Suppose t_1 and t_2 are not disjoint; then terms that are in the intersection can be represented by both t_1 and t_2 and hence the two types will not be distinguishable wherever terms from the intersection can

arise. In effect, a set of disjoint types contains, in such cases, separate types representing $t_1 \cap t_2$, $t_1 \setminus t_2$ and $t_2 \setminus t_1$. In the worst case, it can thus be seen that there is an exponential number of disjoint types for a given set of types. In practice, many of the intersections and set complements are empty and we find usually that the number of disjoint types is similar to the number of given types. Thus with disjoint types, we can obtain a more accurate representation of the set of terms that can appear in a given argument position, while retaining the intuitive, user-oriented notation of arbitrary types. In fact, the type declarations of LOGEN can be used without modification to construct an abstract domain.

The rules for a complete set of disjoint types define a *pre-interpretation* of the signature Σ, whose domain is the set of disjoint types. An abstract interpretation based on this pre-interpretation gives the least model over the domain [2, 3, 12]. This yields success patterns for each program predicate, over the disjoint types. That is, each predicate p/n has a set of possible success patterns $\{p(t_1^1, \ldots, t_n^1); \ldots; p(t_1^m, \ldots, t_n^m)\}$. A set of accurate call patterns can be computed from the model and an initial typed goal. We use the "magic-set" approach to obtain the calls, as described by Codish and Demoen [6]. This yields a set of "call patterns" for each predicate, say $\{p(s_1^1, \ldots, s_n^1); \ldots; p(s_1^k, \ldots, s_n^k)\}$. (Note that we could use a top-down analysis framework, but for analyses based on pre-interpretations, this would give exactly the same results).

Finally the filter for p/n derived from the set of calls is obtained by collecting all the possible types for each argument together. The set of call patterns $\{p(s_1^1, \ldots, s_n^1); \ldots; p(s_1^k, \ldots, s_n^k)\}$ yields the filter $p(\{s_1^1, \ldots, s_1^k\}, \ldots, \{s_n^1, \ldots, s_n^k\})$. For displaying to the user, if required, these filters can be translated back to a description in terms of the original types, rather than the disjoint types.

Analysing Annotated Programs

The standard methods for computing an abstract model and abstract call patterns have to be modified in our procedure, since some body calls may be marked as *memo* or *rescall*. That is, they are not to be unfolded but rather kept in the specialised program. This obviously affects propagation of binding types, since a call annotated as memo or rescall cannot contribute any answer bindings.

When building the abstract model of a program, we simply delete memo-ed and rescall-ed calls from the program, as they cannot contribute anything to the model. Let C be a conjunction of calls; then denote by $\overline{C}$ the conjunction obtained by deleting memo-ed and rescall-ed atoms from C. Let P be an annotated program; then we compute the success patterns for the program $\overline{P} = \{H \leftarrow \overline{B} \mid H \leftarrow B \in P\}$.

When deriving a call pattern, say for atom B_j in clause $H \leftarrow B_1, \ldots, B_j, \ldots$, we ignore the answers to memo-ed and rescall-ed calls occurring in $B_1, \ldots, B_{j-1}$. That is, we consider the clause $H \leftarrow \overline{B_1, \ldots, B_{j-1}}, B_j, \ldots$, when computing the calls to B_j.

4 Termination Checking

Without proper annotations in the source program, the specialiser may fail to terminate. There are two reasons for nontermination:

- **Local Termination:** Unfolding an unsafe call may fail to terminate or provide infinitely many answers.
- **Global Termination:** Even if local termination is ensured, the specialisation may still fail to terminate if it attempts to build infinitely many specialised versions of some predicate for infinitely many different static values.

We do not discuss global termination in this paper. We approach the *local termination* problem using the *binary clause semantics* [9], a representation of a program's computations that makes it possible to reason about loops and hence termination.

Binary Clause Semantics
Informally, the binary clause semantics of a program P is the set of all pairs of atoms (called binary clauses) $p(\bar{X})\theta \leftarrow q(\bar{t})$ such that p is a predicate, $p(\bar{X})$ is a most general atom for p, and there is a finite derivation (with leftmost selection rule) $\leftarrow p(\bar{X}), \ldots, \leftarrow (q(\bar{t}), Q)$ with computed answer substitution θ. In other words a call to $p(\bar{X})$ is followed some time later by a call to $q(\bar{t})$, computing a substitution θ.

We modify the semantics to include *program point* information for each call in the program. A clause $p(ppM, \bar{X})\theta \leftarrow q(ppN, \bar{t})$ details that the call $p(\bar{X})$ at program point ppM is followed sometime later by a call to $q(\bar{t})$ at program point ppN, computing a substitution θ. This extra precision is required to correctly identify the actual unsafe call.

To create the binary clause semantics we specialise a modified vanilla interpreter with respect to our source program. This allows us to easily adapt the semantics for the annotations by changing the rules of the interpreter.

For example, take the classic append program shown in Fig. 3. The transformation to binary clause semantics is shown in Fig. 4. The first clause represents a loop from the call `app([A|B], C, [A|D])` at program point 0 back to itself with the arguments `app(B, C, D)`, the second clause represents an infinite number of possible loops through the same point.

```
app([], B, B).
app([A|As], B, [A|Cs]) :- app(As, B, Cs).
```

Fig. 3. The append program

```
bin_solve_atom__2(0, app([A|B], C, [A|D]), app(B, C, D)).
bin_solve_atom__2(0, app([A|B], C, [A|D]), app(E, F, G)) :-
        bin_solve_atom__2(0, app(B, C, D), app(E, F, G)).
```

Fig. 4. The binary clause version of append from Fig. 3

Convex Hull Abstraction
The binary semantics is in general infinite, but we make a safe approximation of the set of binary clauses using abstract interpretation. We use a domain of convex hulls We use a domain of convex hulls (the convex hull analyser used in our implementation is derived from ones kindly supplied by Genaim and Codish [14]] and by Benoy, King and Mesnard [1]) to abstract the set of binary clauses with respect to a selected norm.

Our implementation currently uses two norms, term size as defined in Eq. 1 and list length as defined in Eq. 2. The use of only two norms effectively restricts ourcurrent implementation to handle only list-processing examples effectively; we are extending the system to derive the norms automatically from the propagated binding types, using techniques described in the literature [17, 27].

$$|t|_{term} = \begin{cases} 1 + \sum_{i=1}^{n} |t_i|_{term} & \text{if } t = f(t_1, ..., t_n) \\ 0 & otherwise \end{cases} \quad (1)$$

$$|t|_{list} = \begin{cases} 1 + |ts|_{list} & \text{if } t = [t|ts] \\ 0 & otherwise \end{cases} \quad (2)$$

Using such an abstraction, we obtain a finite set of binary clauses and a set of constraints representing a linear relationship between the sizes of the respective concrete arguments. Fig. 5 is the binary clause program for append, Fig. 4, abstracted using the domain of convex hulls with respect to the list norm.

```
bin_solve_atom(0, app(A,B,C), app(D,E,F)) :-
        [A = 1 + D, B = E, C = 1 + F,  D > = 0, E >= 0, F >= 0]
```

Fig. 5. Abstract Convex Hull of Fig. 4 using List norm

Checking Termination Criteria
In particular, loops are represented by binary clauses with the same predicate occurring in the head and body. Termination proofs require that for every abstract binary clause between p and p (at the same program point) there is a strict reduction in the size for some *rigid* argument. An argument is rigid if all of its instances have the same size with respect to the selected norm. We detect rigidity by examining the filters derived for the arguments, as illustrated below.

The constraints shown in Fig. 5 show a decrease in the first ($A = 1 + D$) and third argument ($C = 1 + F$). Given the initial filter declaration:

```
:- filter app(type list(dynamic), dynamic, dynamic).
```

The first argument is rigid with respect to the list norm, so termination is proven for this loop providing these binding types. If the filter specified was:

```
:- filter app(dynamic,type list(dynamic), dynamic).
```

Then the call would have to be marked unsafe and would be changed from **unfold** to **memo**, as there is no strict decrease in any rigid arguments.

5 Example

We demonstrate the binding-time analysis using the transpose example shown in Fig. 6. The program takes a matrix, represented as a list of lists, and transposes the rows and columns.

```
/* Created by Pylogen */
/* file: transpose.pl */
transpose(Xs,[]) :-  nullrows(Xs).
transpose(Xs,[Y|Ys]) :-   makerow(Xs,Y,Zs), transpose(Zs,Ys).

makerow([],[],[]).
makerow([[X|Xs]|Ys],[X|Xs1],[Xs|Zs]) :-  makerow(Ys,Xs1,Zs).

nullrows([]).
nullrows([[]|Ns]) :-  nullrows(Ns).
```

Fig. 6. Program for transposing a matrix

The initial filter declaration, providing the binding types of the entry point is `:- filter transpose((type list(dynamic)), dynamic).` The first argument is a list of *dynamic* elements, the length of the list will be known but the individual elements will not be known at specialisation time. The second argument is fully *dynamic*; it will not be given at specialisation time. All calls in the program are initially annotated as `unfold`. Using this initial annotation and the entry types for transpose we propagate the binding types throughout the program. The resultant binding types are shown in Fig. 7. The list structure has been successfully propagated through the clauses of the program.

The next stage of the algorithm looks for possibly non-terminating loops in the annotated program. The result is shown in Fig. 8. The binary clause representation of the program has been abstracted with respect to the list norm over the domain of convex hulls. Termination of each of the loops in Fig. 8 must show a strict decrease in any rigid argument. Based on the propagated binding types only the first argument of each predicate is rigid with respect to the list norm. The predicate `makerow/3` has a strict decrease (`A=1.0+D`), `nullrows/1` also has a strict decrease (`A=1.0+B`) but the recursive call to transpose has no decrease in a rigid argument and is unsafe.

```
:- filter   transpose((type list(dynamic)), dynamic).
:- filter   makerow((type list(dynamic)), dynamic, dynamic).
:- filter   nullrows((type list(dynamic))).
```

Fig. 7. Propagated filters for Fig. 6 using the initial filter `transpose((type list(dynamic)), dynamic)`

```
bin_solve_atom(3, makerow(A,B,C), makerow(D,E,F)) :-
        [A=1.0+D,D>=0.0,B=1.0+E,E>=0.0,C=1.0+F,F>=0.0].
bin_solve_atom(4, nullrows(A), nullrows(B)) :-
        [A=1.0+B,B>=0.0].
bin_solve_atom(2, transpose(A,B), transpose(C,D)) :-
        [B>D,C>=0.0,D>=0.0,A=C,B=1.0+D].

%% Loop at program point 2 is unsafe (transpose/2)
```

Fig. 8. Binary clause representation of Fig.6 abstracted over the domain of convex hulls with respect to the list norm

```
logen(transpose, transpose(A,[])) :-
        logen(unfold, nullrows(A)).
logen(transpose, transpose(A,[B|C])) :-
        logen(unfold, makerow(A,B,D)),
        logen(memo, transpose(D,C)).
logen(makerow, makerow([],[],[])).
logen(makerow, makerow([[A|B]|C],[A|D],[B|E])) :-
        logen(unfold, makerow(C,D,E)).
logen(nullrows, nullrows([])).
logen(nullrows, nullrows([[]|A])) :-
        logen(unfold, nullrows(A)).

:- filter    makerow((type list(dynamic)), dynamic, dynamic).
:- filter    nullrows((type list(dynamic))).
:- filter    transpose((type list(dynamic)), dynamic).
```

Fig. 9. Annotated version of Transpose from Fig. 6

Marking the offending unsafe call as *memo* removes the potential loop and further iterations through the algorithm produce no additional unsafe calls. The final output of the BTA algorithm is shown in Fig. 9.

6 Experimental Results

The automatic binding-time analysis detailed in this paper is implemented as part of the LOGEN partial evaluation system. The system has been tested using benchmarks derived from the DPPD benchmark library [18]. The figures in Table 1 present the timing results[1] from running the BTA on an unmodified program given an initial filter declaration. These benchmark examples along

[1] The execution time for the Original and Specialised code is based on executing the benchmark query 20,000 times on a 2.4Ghz Pentium with 512Mb running SICStus Prolog 3.11.1. The specialisation times for all examples was under 10ms.

Table 1. Benchmark Figures for Automatic Binding-Time Analysis

Benchmark	BTA	Original	Specialised	Relative Time
combined	3220ms	110ms	30ms	0.27
inter binding	1380ms	60ms	10ms	0.17
inter medium	1440ms	140ms	10ms	0.07
inter simple	2670ms	80ms	30ms	0.38
match	400ms	90ms	70ms	0.78
regexp	780ms	220ms	60ms	0.28
transpose	510ms	80ms	10ms	0.13

with the PYLOGEN system, shown in Fig. 10, can be downloaded from LOGEN website[2].

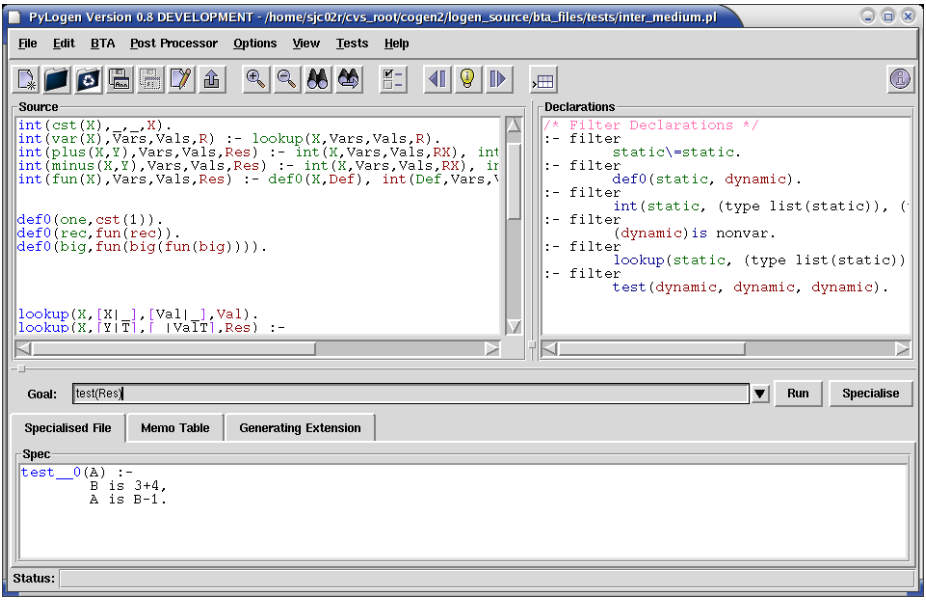

Fig. 10. Snapshot of a PYLOGEN session

- **combined** - A test case combining the inter medium, inter simple and regular expression interpreters.
- **inter binding** - An interpreter using a partially static data structure for an environment. In this example we combine the list and term norms.

[2] http://www.asap.soton.ac.uk/logen

- **inter medium** - An interpreter with the environment split into two separate lists, one for the static names the other for the dynamic values.
- **inter simple** - A simple interpreter with no environment, but contains a selection of built-in arithmetic functions.
- **match** - A string pattern matcher.
- **regexp** - An interpreter for regular expressions.
- **transpose** - A matrix transpose program.

7 Related Work and Conclusion

To the best of our knowledge, the first binding-time analysis for logic programming is [5]. The approach of [5] obtains the required annotations by analysing the behaviour of an *online* specialiser on the subject program. Unfortunately, the approach was overly conservative. Indeed, [5] decides whether or not to unfold a call based on the original program, not taking current annotations into account. This means that a call can either be completely unfolded or not at all. Also, the approach was never fully implemented and integrated into a partial evaluator.

In Section 6 of [20] a more precise BTA has been presented, which has been partially implemented. It is actually the precursor of the BTA here. However, the approach was not fully implemented and did not consider the issue of filter propagation (filters were supposed to be correct). Also, the identification of unsafe calls was less precise as it did not use the binary clause semantics with program points (i.e., calls may have been classified as unsafe even though they were not part of a loop).

[26] is probably the most closely related work to ours. This work has a lot in common with ours, and we were unaware of this work while developing our present work.[3] Let us point out the differences. Similar to [20], [26] was not fully implemented (as far as we know, based on the outcome of the termination analysis, the user still had to manually update the annotations by hand) and also did not consider the issue of filter propagation. Also, [26] cannot handle the `nonvar` annotation (this means that, e.g., it can only handle the vanilla interpreter if the call to the object program is fully static). However, contrary to [20], and similar to our approach, [26] does use the binary clause semantics. It even uses program point information to identify non-terminating calls. However, we have gone one step further in using program point information, as we will only look for loops from one program point back to itself. Take for example the following program:

```
p(a) :- q(a).          q(a) :- q(b).          q(b) :- q(b).
```

Both our approach and [26] will mark the call `q(a)` as unfoldable and the call `q(b)` in clause 3 as unsafe. However, due to the additional use of program points,

[3] Thanks for reviewers of LOPSTR'04 for pointing this work out to us.

we are able to mark the call `q(b)` in clause 2 as unfoldable (as there is no loop from that program point back to itself), whereas we believe that [26] will mark it as unsafe. We believe that this extra precision may pay off for interpreters. Finally, due to the use of our meta-programming approach we can handle the full LOGEN annotations (such as `call`, `rescall`, `resif`,...) and can adapt our approach to compute memoisation loops and tackle global termination.

The papers [24, 25, 28] describe various BTAs for Mercury, even addressing issues such as modularity and higher-order predicates. An essential part of these approaches is the classification of unifications (using Mercury's type and mode information) into tests, assignments, constructions and deconstructions. Hence, these works cannot be easily ported to a Prolog setting, although some ideas can be found in [28].

Currently our implementation guarantees correctness and termination at the local level, and correctness but not yet termination at the global level. However, the framework can very easily be extended to ensure global termination as well. Indeed, our binary clause interpreter can also compute memoisation loops, and so we can apply exactly the same procedure as for local termination. Then, if a memoised call is detected to be unsafe we have to mark the non-decreasing arguments as dynamic. Finally, as has been shown in [11], one can actually relax the strict decrease requirement for global termination (i.e., one can use $\leq$ rather than $<$), provided so-called "finitely partitioning" norms are used.

Acknowledgements. Thanks to Dan Elphick for his work on the Python mode for Logen.

References

1. F. Benoy, A. King, and F. Mesnard. Computing Convex Hulls with a Linear Solver. *Theory and Practice of Logic Programming*, January 2004.
2. D. Boulanger and M. Bruynooghe. A systematic construction of abstract domains. In B. Le Charlier, editor, *Proc. First International Static Analysis Symposium, SAS'94*, volume 864 of *Springer-Verlag Lecture Notes in Computer Science*, pages 61–77, 1994.
3. D. Boulanger, M. Bruynooghe, and M. Denecker. Abstracting *s*-semantics using a model-theoretic approach. In M. Hermenegildo and J. Penjam, editors, *Proc. 6th International Symposium on Programming Language Implementation and Logic Programming, PLILP'94*, volume 844 of *Springer-Verlag Lecture Notes in Computer Science*, pages 432–446, 1994.
4. M. Bruynooghe and G. Janssens. An instance of abstract interpretation integrating type and mode inferencing. In R. Kowalski and K. Bowen, editors, *Proceedings of ICLP/SLP*, pages 669–683. MIT Press, 1988.
5. M. Bruynooghe, M. Leuschel, and K. Sagonas. A polyvariant binding-time analysis for off-line partial deduction. In C. Hankin, editor, *Proceedings of the European Symposium on Programming (ESOP'98)*, LNCS 1381, pages 27–41. Springer-Verlag, April 1998.

6. M. Codish and B. Demoen. Analysing logic programs using "Prop"-ositional logic programs and a magic wand. In D. Miller, editor, *Proceedings of the 1993 International Symposium on Logic Programming, Vancouver*. MIT Press, 1993.
7. M. Codish and B. Demoen. Deriving type dependencies for logic programs using multiple incarnations of Prop. In B. Le Charlier, editor, *Proceedings of SAS'94, Namur, Belgium*, 1994.
8. M. Codish and V. Lagoon. Type dependencies for logic programs using ACI-unification. *Theoretical Computer Science*, 238(1-2):131–159, 2000.
9. M. Codish and C. Taboch. A semantic basic for the termination analysis of logic programs. *The Journal of Logic Programming*, 41(1):103–123, 1999.
10. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications*. http://www.grappa.univ-lille3.fr/tata, 1999.
11. S. Decorte, D. De Schreye, M. Leuschel, B. Martens, and K. Sagonas. Termination analysis for tabled logic programming. In N. Fuchs, editor, *Proceedings of the International Workshop on Logic Program Synthesis and Transformation (LOPSTR'97)*, LNCS 1463, pages 111–127, Leuven, Belgium, July 1998.
12. J. Gallagher, D. Boulanger, and H. Sağlam. Practical model-based static analysis for definite logic programs. In J. W. Lloyd, editor, *Proc. of International Logic Programming Symposium*, pages 351–365, 1995.
13. J. P. Gallagher and K. Henriksen. Abstract domains based on regular types. In V. Lifschitz and B. Demoen, editors, *Proceedings of the International Conference on Logic Programming (ICLP'2004)*, volume 3132 of *LNCS*. Springer Verlag, 2004.
14. S. Genaim and M. Codish. Inferring termination conditions of logic programs by backwards analysis. In *International Conference on Logic for Programming, Artificial intelligence and reasoning*, volume 2250 of *Springer Lecture Notes in Artificial Intelligence*, pages 681–690, 2001.
15. P. Hill and J. Gallagher. Meta-programming in logic programming. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, pages 421–497. Oxford Science Publications, Oxford University Press, 1998.
16. K. Horiuchi and T. Kanamori. Polymorphic type inference in prolog by abstract interpretation. In *Proc. 6th Conference on Logic Programming*, volume 315 of *Springer-Verlag Lecture Notes in Computer Science*, pages 195–214, 1987.
17. V. Lagoon, F. Mesnard, and P. J. Stuckey. Termination analysis with types is more accurate. In C. Palamidessi, editor, *Proceedings of Logic Programming, 19th International Conference, ICLP 2003*, volume 2916 of *Springer-Verlag Lecture Notes in Computer Science*, pages 254–268, 2003.
18. M. Leuschel. The ECCE partial deduction system and the DPPD library of benchmarks. Obtainable via `http://www.ecs.soton.ac.uk/~mal`, 1996-2002.
19. M. Leuschel, S.-J. Craig, M. Bruynooghe, and W. Vanhoof. Specializing interpreters using offline partial deduction. In M. Bruynooghe and K.-K. Lau, editors, *Program Development in Computational Logic*, LNCS 3049, pages 341–376. Springer-Verlag, 2004.
20. M. Leuschel, J. Jørgensen, W. Vanhoof, and M. Bruynooghe. Offline specialisation in Prolog using a hand-written compiler generator. *Theory and Practice of Logic Programming*, 4(1):139–191, 2004.
21. B. Martens and D. De Schreye. Two semantics for definite meta-programs, using the non-ground representation. In K. R. Apt and F. Turini, editors, *Meta-logics and Logic Programming*, pages 57–82. MIT Press, 1995.

22. 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.
23. 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.
24. W. Vanhoof. Binding-time analysis by constraint solving: a modular and higher-order approach for Mercury. In M. Parigot and A. Voronkov, editors, *Proceedings of LPAR'2000*, LNAI 1955, pages 399–416. Springer-Verlag, 2000.
25. W. Vanhoof and M. Bruynooghe. Binding-time analysis for Mercury. In D. De Schreye, editor, *Proceedings of the International Conference on Logic Programming ICLP'99*, pages 500–514. MIT Press, 1999.
26. W. Vanhoof and M. Bruynooghe. Binding-time annotations without binding-time analysis. In R. Nieuwenhuis and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 8th International Conference*, LNCS 2250, pages 707–722. Springer-Verlag, 2001.
27. W. Vanhoof and M. Bruynooghe. When size does matter. In A. Pettorossi, editor, *Logic Based Program Synthesis and Transformation, 11th International Workshop, LOPSTR 2001*, Springer-Verlag Lecture Notes in Computer Science, pages 129–.147, 2001.
28. W. Vanhoof, M. Bruynooghe, and M. Leuschel. Binding-time analysis for Mercury. In M. Bruynooghe and K.-K. Lau, editors, *Program Development in Computational Logic*, LNCS this Volume. Springer-Verlag, 2004.