Part I

Framework for A Fully Automatic Binding Time Analysis

1 Offline Partial Evaluation

Most offline approaches perform what is called a *binding-time analysis (*BTA*)* prior to the specialization phase. The partial evaluator uses the generated annotated program to guide the specialization process. There are two types of annotations:

- **filter declarations**, indicating whether arguments to predicates are **static**, **dynamic**, or possibly some other binding type. These annotations influence the generalization of goals and thus the global control.

- **clause annotations**, indicating how every call in the body should be treated during unfolding. These influence the local control.

In this paper we outline a procedure for generating these annotations automatically. There are several independent components of the procedure. Hence, if one of these components is missing it could be replaced by a manual procedure or oracle, making the whole process semi-automatic.

The input to the procedure is (i) a program to be specialized, (ii) a set of types, and (iii) a goal whose arguments are typed with respect to the given types. Note that the program need not be a typed program in the usual sense. The types simply define sets of terms which form the basis for the binding type annotations.

The components of the automatic BTA are (i) a type determinization algorithm, (ii) an abstract interpreter over a domain of determinized (disjoint) types, and (iii) a termination analyser, incorporating for example an abstract interpreter over a domain of convex hulls.

1.1 Binding Types

The basis of the BTA is a classification of arguments using abstract values. Abstract values can be chosen in different ways. The BTA procedure described here is independent of the particular choice of abstraction; the only essential aspect of the abstraction is that it must be possible to determine whether an argument is possibly unbound (that is, it is could be a variable). In this case, the argument is called *dynamic*.

The simplest approach is to classify arguments within the program to be specialized as either *static* or *dynamic*. The value of a static argument will be *definitely ground*. A dynamic argument can be either ground or not ground, that is, it can be any term. The use of static-dynamic binding types was introduced for r functional programs, and has been used in logic program BTAs [**?**]. While often sufficient for functional programs, it sometimes proves to be too weak for logic programs: in logic programming partially instantiated data structures appear naturally even at runtime. A simple classification of arguments into "fully known" or "totally unknown" is therefore unsatisfactory and would prevent specialising a lot of "natural" logic programs such as the vanilla metainterpreter [**?**, **?**] or most of the benchmarks from the DPPD library [**?**].

Other more expressive binding types can be based on lists or other datatypes. Using typical notation for defining types, one can define lists (or trees etc.) whose elements are ground, dynamic, numbers, nested structures and so on. These types must be representable as a set of regular types rules. The sets of ground and non-ground terms can also be defined as regular types. Examples are given in [**?**].

1.1.1 Derivation of Filter Declarations

The given set of types is first transformed into a set of disjoint regular types. This process is called *determinization* and is described in detail in another report [**?**]. The disjoint types define an abstract domain, which is input, along with the typed goal, to an abstract interpreter over that domain. The details of the abstract interpretation algorithm are given in [**?**].

For example, the simple binding types *static* and *dynamic* are determinized to the disjoint types *static* and *non-static*. Note that *dynamic* is the union of *static* and *non-static*.

The output of the abstract interpreter is an assignment of a set of disjoint

types to each argument. Let p/n be a predicate in the program. Then the analyser assigns a binding type $p(S_1, \ldots, S_n)$ to p, where S_j $(1 \leq j \leq n)$ is a set of disjoint types $\{t_{j_1}, \ldots, t_{j_k}\}$. The meaning of the assignment is that, whenever p/n is called during specialization, its j^{th} argument has a value given by one of the types $\{t_{j_1}, \ldots, t_{j_k}\}$.

The assignment of types to arguments for a predicate p is called a *filter* for p. The filter is used in the generalization operation dueing partial evalaution. We assume that there is a generalization operation, which takes as input an atom for predicate p and a filter for p, and returns another, more general atom for p.

1.2 Clause Annotations

Each clause is annotated indicating how every call in the body should be treated during specialization. This is needed to ensure local termination (that is, the termination of the unfolding for each atom). Some of the basic annotations are:

- **unfold** for reducible predicates: they will be unfolded during specialization.
- **memo** for non-reducible predicates: they will be added to the memoisation table and replaced with a generalized residual predicate (using the filter for that predicate).
- **call** fully static call will be completely evaluated during specialization.
- **rescall** the call will be kept and will appear in the final specialised code.

1.3 Example Annotation

The simple annotations are demonstrated using the **Logen** annotation syntax:

```
:- filter append(static, dynamic, dynamic).
append([],L,L).
append([H|T],L,[H|T1]) :- logen(unfold, append(T,L,T1).
```

The first argument to append has been marked as **static**, it will be known at specialization time, and the other arguments have been marked **dynamic**. Note that the filter declaration above could be derived automatically from the append program, the types *static* and *dynamic*, and the typed goal *append(static, dynamic, dynamic)*.

After determinizing the types, the filter would be represented in the following equivalent form.

```
:-filter append([static],[static,nonstatic],[static,nonstatic]).
```

In other words, the second and third argument can be either static or nonstatic, which is the same as dynamic. Note that the analysis guarantees correct filters. In a manually written filter, it is not guaranteed that calls to *append* would actually satisfy the given binding types (e.g. that *append* would always be called with a ground first argument).

Examining the clause annotation in the example, the recursive call to append is annotated for unfolding. The first argument is static and this is sufficient to guarantee termination at specialization time. If we could not guarantee the termination of the recursive call then it would have to be marked as **memo**.

1.3.1 Derivation of Filter Declarations in the Presence of Clause Annotations

In the derivation of filter declarations mentioned above it was stated that whenever p/n is called during specialization, its j^{th} argument has a value given by one of the types $\{t_{j_1}, \ldots, t_{j_k}\}$. It was not explained what is meant by "whenever p/n is called during specialization". In fact, this depends on the clause annotations, since the decision of whether to memo or call an atom affects the propagation of binding types.

Standard abstract interpretation, which assumes a complete computation, is modified to allow for memo-ing of calls. Memo-ed called are simply ignored when propagating binding types. The same holds for *rescall* annotations. Note that we still derive filters for memo-ed calls: it is only for propagation of binding types that they are ignored.

2 Outline of Algorithm

We now outline the main operations and steps in the algorithm. The core of the algorithm is a loop which propagates the binding types with respect to the current clause annotations, checks the resulting filters for termination conditions, and modifies the clause annotations accordingly. Initiially, all body atoms are annotated as *unfold* or *call*. These can be changed to *memo* or *rescall*, until termination is established.

1. **Initialisation:**
   (a) **getProgramTypes()** Standard (static, dynamic, list(dynamic), var) together with user types from type inference and user declaration.
   (b) **determinizeTypes()** Transform types into disjoint intersection types (DIT).
   (c) **initialiseAnnotations()** Mark everything as unfolded and add a type declaration for the initial query.
2. **Type/Filter inference**
   (a) **removeMemoedCalls()** Remove all memoed calls for the analysis
   (b) **bottomUpAnalysis()** With Respect to DIT types. We get a precise type inferred for every predicate
   (c) **for** all filtered calls **do**
   propagateTypes() - using the result of bottom up analysis. We get call types for every program point
   end for
3. **Local Unfolding Problem**
   (a) **createUnfoldBinaryClauseSemantics()** Partially Evaluate bin_solve, specialise with respect to filter goals and the annotated program. Producing a program whose semantics is the binary clause semantics for local unfoldings
   (b) **abstractProgram()** Abstract binaryClause program wrt some norm(s) using the types inferred earlier

(c) **convexHullAnalysis()** Run (Andy King's) Convex Hull Analyser on abstract program

(d) **for** each binary clause C of the form *bin_solve(p(–),p(–), pp)* **do**

if (a rigid argument of C decreases) **then** local termination is ensured

else mark offending call point(s) as memo and restart at 2

endif

endfor

4. **Global Termination Problem**

(a) **createMemoBinaryClauseSemantics()** Partially Evaluate bin_solve to produce binary clause semantics for the memoed calls

(b) **naïve approach:** Use standard termination analysis with two changes:

i. ensure that every static part of the argument is measured (finitely partitioning condition)

ii. only need to use $=<$ instead of $<$ on all binary recursive calls

3 Binary Clause Semantics

`Bin_solve` is a binary clause interpreter based on the simple vanilla interpreter solve. Specialising bin_solve with respect to the filter goals and annotated programs we wish to analyse produces a program whose semantics is the binary clause semantics for either local unfolding or memoed calls.

```
:- filter solve(type(list(nonvar))):solve.
bin_solve([unfold(H)|_T],H).
bin_solve([memo(H)|_T],memo(H)).
bin_solve([unfold(H)|_T],RecCall) :-
        bin_solve_atom(H,RecCall).
bin_solve([unfold(H)|T],RecCall) :-
        solve_atom(H),
        bin_solve(T,RecCall).
bin_solve([memo(_)|T],RecCall) :-
        bin_solve(T,RecCall).
```

```
:- filter bin_solve_atom(nonvar_nf,dynamic).
bin_solve_atom(H,Rec) :-
            rule(H,Bdy),
                bin_solve(Bdy,Rec).

:- filter test(dynamic,dynamic).
test(H,Rec) :-
                filtered(H),
                bin_solve([unfold(H)],Rec).

filtered(ann_dapp(_,_,_,_)).
filtered(ann_app(_,_,_)).

:- module('/home/sjc02r/cvs_root/cogen2/logen_examples/bta_test/bin_solve.pl.memo'
gensym(num__num,5).
memo_table(0,test(A,B),test__0(A,B),done(user),[]).
memo_table(1,bin_solve_atom(ann_dapp(A,B,C,D),E),bin_solve_atom__1(ann_dapp(A,B,C,
memo_table(2,solve_atom(ann_dapp(A,B,C,D)),solve_atom__2(A,B,C,D),done(internal),[
memo_table(3,bin_solve_atom(ann_app(A,B,C),D),bin_solve_atom__3(ann_app(A,B,C),D),
memo_table(4,solve_atom(ann_app(A,B,C)),solve_atom__4(A,B,C),done(internal),[]).

test__0(ann_dapp(B,C,D,E),ann_dapp(B,C,D,E)).
test__0(ann_dapp(B,C,D,E),F) :-
  bin_solve_atom__1(ann_dapp(B,C,D,E),F).
test__0(ann_app(B,C,D),ann_app(B,C,D)).
test__0(ann_app(B,C,D),E) :-
  bin_solve_atom__3(ann_app(B,C,D),E).
bin_solve_atom__1(ann_dapp(B,C,D,E),ann_app(C,D,F)).
bin_solve_atom__1(ann_dapp(B,C,D,E),F) :-
  bin_solve_atom__3(ann_app(C,D,G),F).
bin_solve_atom__1(ann_dapp(B,C,D,E),memo(app(B,F,E))) :-
  solve_atom__4(C,D,F).
solve_atom__2(B,C,D,E) :-
  solve_atom__4(C,D,F).
bin_solve_atom__3(ann_app([B|C],D,[B|E]),memo(ann_app(C,D,E))).
solve_atom__4([],B,B).
solve_atom__4([B|C],D,[B|E]).
```

4 Worked Example

We demonstrate the alogrithm using a worked example of a pattern matching program.

The `match` predicate identifies a pattern `Pat` in a string `T`.

```
match(Pat,T) :-
        match1(Pat,T,Pat,T).
match1([],Ts,P,T).
match1([A|Ps],[B|Ts],P,[X|T]) :-
        A\==B,
        match1(P,T,P,T).
match1([A|Ps],[A|Ts],P,T) :-
        match1(Ps,Ts,P,T).
```

Initialise the annotations to all unfold and call?

```
:- filter match(list, dynamic).
match(Pat,T) :-
        unfold(match1(Pat,T,Pat,T)).
match1([],Ts,P,T).
match1([A|Ps],[B|Ts],P,[X|T]) :-
        call(A\==B),
        unfold(match1(P,T,P,T)).
match1([A|Ps],[A|Ts],P,T) :-
        unfold(match1(Ps,Ts,P,T)).
```

Specialising through bin_solve to obtain the abstract clause semantics:

```
bin_solve_atom__0(match(B,C),unfold(match1(B,C,B,C))).
bin_solve_atom__1(match1([B|C],[D|E],F,[G|H]),unfold(match1(F,H,F,H))).
bin_solve_atom__1(match1([B|C],[B|D],E,F),unfold(match1(C,D,E,F))).
```

Abstracting the programming with respect to list length norm

```
:- filter match(list,dynamic).
:- filter match1(list,dynamic,list,dynamic).
bin_solve_atom__0(match(B,_),unfold(match1(B,_,B,_))).
bin_solve_atom__1(match1(1+C,_,F,_),unfold(match1(F,_,F,_))).
bin_solve_atom__1(match1(1+C,_,E,_),unfold(match1(C,_,E,_))).
```

For each binary clause of the form: `bin_solve_atom(p(--),unfold(p(--))`:

```
bin_solve_atom__1(match1(1+C,_,F,_),unfold(match1(F,_,F,_))).
```

Argument 1 does not show decrease 1+C → F
Argument 3 does not show decrease F → F
Offending call must be marked as memo.

```
:- filter match(list, dynamic).
match(Pat,T) :-
        unfold(match1(Pat,T,Pat,T)).
match1([],Ts,P,T).
match1([A|Ps],[B|Ts],P,[X|T]) :-
        call(A\==B),
        memo(match1(P,T,P,T)).
match1([A|Ps],[A|Ts],P,T) :-
        unfold(match1(Ps,Ts,P,T)).

bin_solve_atom__0(match(B,C),unfold(match1(B,C,B,C))).
bin_solve_atom__1(match1([B|C],[B|D],E,F),unfold(match1(C,D,E,F))).
```

Abstract program again ...

```
bin_solve_atom__0(match(B,C),unfold(match1(B,C,B,C))).
bin_solve_atom__1(match1(1 +C,_,E,_),unfold(match1(C,_,E,_))).
```

For each binary clause of the form: `bin_solve_atom(p(--),unfold(p(--))`:

```
bin_solve_atom__1(match1(1 +C,_,E,_),unfold(match1(C,_,E,_))).
```

Argument 1 does show decrease 1+C → C
Local termination is ensured.