# Automatic Binding Time Analysis Algorithm

John P. Gallagher Kim S. Henriksen

January 15, 2004

## 1 Introduction to the BTA problem

The problem is as follows. Given a logic program and a goal or set of goals described by a pattern, type etc, decide on an unfolding rule for offline partial evaluation.

An unfolding rule, in this context, is defined by an annotation of the program. Each body atom of the program is marked as follows:

- *memo*: do not select the atom for unfolding.
- *unfold*: the atom is selected for unfolding whenever it is the leftmost selectable atom in the resolvent.
- *call*: the atom is selected for unfolding whenever it is the leftmost selectable atom in the resolvent, and the complete computation tree is constructed (that is, the atom is completely evaluated). *rescall*: do not select the atom for unfolding. This is no different from *memo* as regards unfolding, but there is a difference in code generation, since atoms marked *rescall* are assumed to be defined outside the program.

The main decision to make is therefore between selecting or not selecting each atom (*memo* or *rescall* versus *unfold* or *call*.

For a given atom, the choice of annotation depends on two main considerations.

- The state of instantiation of the atom when it is potentially selectable.
- The need to ensure termination of unfolding.

The first consideration can be solved by fairly standard static analysis of the program with respect to the given goal(s). The second requires an analysis of the termination behaviour of the program itself (to ensure termination of unfolding) and also the termination behaviour of the partial evaluation algorithm (to ensure global termination).

## 2 Propagating Call and Success Patterns

We use the analysis framework described in [?] and [?]. This consists of the following steps:

- Abstract a program to a *domain program*, in which every non-variable term $f(x_1, \ldots, x_n)$ in a clause (starting with innermost terms) is replaced by a fresh variable $u$, and an atom $denotes(f(x_1, \ldots, x_n), u)$ is added to the clause body.
- Supply a definition of the *denotes* predicate. This is formally a pre-interpretation of the function symbols of the program over an abstract domain $D$. It can also be seen as a complete bottom-up deterministic tree automaton in which the set of states is $D$, and the *denotes* predicate gives the transitions.
- The bottom-up step: Compute the least model of the transformed program together with the *denotes* program. This yields a set of *success patterns* over the abstract domain $D$.
- The top-down step: Compute call patterns by using a magic-set transformation, with respect to the given goal abstracted in the domain $D$. The answer predicates in the magic-set program are the success patterns obtained from the previous step. Note that the abstract domain defined by a *denotes* predicate is *condensing*, which means that the call patterns computed in this way are as precise as those derived from a goal-directed analysis.
- Use the call patterns to derive a call pattern for each body atom, as well as a single call pattern for each predicate.

## 3 Defining the Abstract Domain From Regular Types

We have developed a novel technique for defining an abstract domain based on regular types. Given some regular types, a *determinisation* procedure is applied. Following this a *completion* is derived. The result is a definition of types which are (a) disjoint and (b) recognise every term in the given signature (the set of functions occurring in the program and goals).

To be described in detail by Kim.....

Typically we start from some standard regular types. These are the set of ground terms, and the set of all terms. These sets are not normally thought of as types. A novel aspect of our approach is that it allows mode information to be integrated seamlessly with type information.

The type consisting of ground terms is called *static*. The set of all terms, called *dynamic* including both ground and non-ground terms, is realised by adding a special constant $v$ with type $var$. $v$ also has the type *dynamic*. The constant $v$ does not appear in any program or goal. Hence, the only way it can appear in the argument of a call or success pattern is that if that argument is possibly variable. The use of such constants, called *no-term elements* in [?] is a way of tracking freeness information in a model-based, declarative analysis.

The standard types are as follows.

```
a -> static.
[] -> static.
[static|static] -> static.
% f(static,static,....,static) -> static  for each f

[] -> list.
[dynamic|list] -> list.

v -> var.

v -> dynamic.
a -> dynamic.
[] -> dynamic.
[dynamic|dynamic] -> dynamic.
% f(dynamic,dynamic,....,dynamic) -> dynamic  for each f
```

User defined types can then be added to these before the determinsation process. Note that the presence of the types *static* and *dynamic* ensures that the determinised automaton is also complete.

## 4 Derivation of Filters

The filters are simply the call patterns for each predicate.

## 5 Analysis of Annotated Programs

The process of deriving annotations is an iterative one. Starting from a program in which every call is marked as unfold, we derive call patterns and filters. Some of the unfold annotations are then changed to *memo* (or *rescall* if the derived call pattern is considered too general to be unfolded. Then the analysis is re-run.

Each atom marked at memo is simply ignored when computing the abstract model of the program (the bottom-up step). Memo-ed atoms are simply omitted, since they contribute no answers.

In the top-down step, the "query clauses" in the magic set transformation are modified so that the answer atoms for memo-ed atoms are omitted.