Towards modular binding-time analysis for first-order Mercury

Vanhoof, Wim; Bruynooghe, Maurice

Published in:
WOID'99, Workshop on Optimization and Implementation of Declarative Programs (in connection with ICLP'99, International Conference on Logic Programming)

DOI:
10.1016/S1571-0661(05)80639-5

Publication date:
1999

Document Version
Publisher's PDF, also known as Version of record

URL: http://www.elsevier.nl/locate/entcs/volume30.html 10 pages

Towards Modular Binding-Time Analysis for First-order Mercury

Wim Vanhoof [1] Maurice Bruynooghe [2]

Department of Computer Science,
K.U.Leuven, Belgium

Abstract

In this paper, we describe work in progress on binding-time analysis (BTA) for a first-order subset of Mercury. BTA is the core part of any off-line specialisation system. We formulate BTA by constraint normalisation, enabling the analysis to be performed efficiently and in a modular way.

1 Introduction and Motivation

Recently, Mercury was introduced as a logic programming language, specifically tuned towards the creation of large-scale, real-world applications [2]. When writing large applications, the programmer usually is encouraged to write *general* code, that can be used in a number of different situations, and to *abstract*, for example concrete data representations by hiding the representation behind a number of procedure calls. To support the programmer employing such software engineering capabilities, Mercury provides a system of type-, mode- and determinism declarations and a flexible module system.

Employing abstraction and generality, however, imposes a penalty on the efficiency of the resulting program, due to the presence of e.g. extra procedure calls and tests with (partially) known input. Program specialisation is a source-to-source transformation, capable of removing precisely these kinds of inefficiencies from a program, by *specialising* general routines with respect to the specific context they are used in, as such concretising the code and removing layers of abstraction. Specialisation is achieved by performing, at specialisation-time, those computations for which enough input is already available [1].

[1] Supported by a specialisation grant of the Flemish Insitute for the Promotion of Scientific-Technological Research in Industry (IWT)
[2] Supported by "FWO Vlaanderen"

In the off-line approach to specialisation [1], the program that is to be specialised, say P, is first analysed by a so-called binding-time analysis (BTA): Given P, an initial call and the instantiatedness of that call's arguments (i.e. a description of how much of their value is known at specialisation-time), BTA computes, for each program variable, its instantiatedness at specialisation-time. Using this information, a number of instructions are generated, specifying what goals should be *reduced* (evaluated during specialisation) or *residualised* (recorded in the residual program). The actual specialisation is performed afterwards, by simply following the instructions generated by BTA.

Binding-time analysis can straightforwardly be described as an application of top-down, call dependent abstract interpretation: starting from the initial call and binding-times for its arguments, the body of the called predicate is analysed resulting in newly computed binding-times. When a predicate call is encountered during analysis, a fresh such analysis is performed for the called predicate using the binding-times from the call.

In recent work [3], we showed the applicability of BTA for Mercury and described a BTA by abstract interpretation. However, such a call dependent analysis imposes some problems when it is used for analysing a multi-module program. In Mercury (as in other languages), a module M exports a number of declarations (e.g. types, predicates,...) that can be used (imported) in another module M'. These modules can be compiled separately: once M is compiled, M' can be compiled over and over again, without the need to recompile M. Consider a program P consisting of the modules $M_1, \ldots, M_n$. The following issues rise when one wants to perform BTA for P using top-down abstract interpretation:

- Since the analysis is performed top-down (p's body is analysed when a call to p is encountered), the complete source of $M_1, \ldots, M_n$ must be available to the analysis.
- A fresh data flow analysis of the same predicate is performed every time a call to this predicate (with a different call pattern) is encountered.
- If M is a module that is used in a number of different programs, the called predicates of M are re-analysed in every program M is used in.

In this work, we formulate a binding-time analysis for Mercury using a framework of constraint normalisation. The main contributions of this work are that the proposed BTA is capable of dealing with predicate modes other than simply `in` and `out` (a restriction present in [3]) and the fact that a large part of the present analysis is call-independent, allowing this part to be performed in a modular way.

The remainder of the paper is organised as follows: in its main body (Section 2), we informally introduce some necessary concepts and notation, and present how binding-time analysis can be performed in a three-stage process. Section 3 concludes with some discussion and directions for future work.

2 Towards Modular Binding-Time Analysis

2.1 Mercury Preliminaries

Mercury is a statically typed logic language, in which the (possibly polymorphic) type of every program variable is statically known. These types are traditionally represented through finite type graphs having two kinds of nodes: a type is represented by a type node, having, for each of the functors of its definition, a functor node which in turn has a number of type nodes as children: one for each argument of the functor. Given a type graph for type t, a type node t' in the graph can be uniquely defined by its *type path*, which is a sequence over functor/integer pairs, describing the path in the graph from t to t'. The set of all such type paths is denoted by $TPath$.

An *instantiatedness graph* is a type graph in which every type node is labelled *bound* or *free* with the constraint that all descendants of a free node must be free.

Example 2.1 Consider the following definition of a type $list(T)$: `list(T) ---> [] ; [T | list(T)]`. Figure 1 depicts the tree possible instantiatedness graphs of the underlying type graph for $list(T)$.

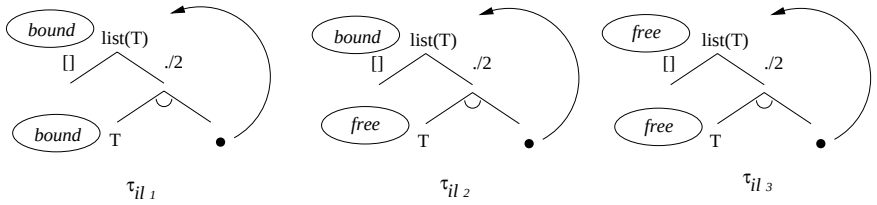

Fig. 1. Instantiatedness graphs for $list(T)$

In order to compare instantiatedness graphs, we introduce the boolean domain $\mathcal{B} = (\{bound, free\}, >)$ on which the order $free > bound$ is imposed. Instantiatedness graphs are traditionally used in Mercury to define the *mode* of a predicate: for each argument, the programmer defines a mapping from an *initial* instantiatedness graph (describing this argument's instantiation at predicate entry) to a *final* instantiatedness graph (describing its instantiation at predicate exit).

Example 2.2 Consider the following predicate definition for *append*, where we use the abbreviations `free` for a type graph in which all nodes are labelled $free$, and `ground` for type graph in which all nodes are labelled *bound*.

```
:- pred append(list(T)::ground -> ground,
               list(T)::ground -> ground,
               list(T)::free -> ground).
```

The definition says that the three arguments of *append* are of type *list*, and that the first two will be completely ground at procedure entry (input), but the

third will be free at procedure entry and ground at procedure exit (output).

In the remainder of this paper, we consider Mercury modules that are in superhomogeneous form [2]. Such a module consists of a number of procedure definitions, a procedure being a predicate with precisely one mode declaration. The definition of a procedure is given by a single clause: the arguments in the head of the clause and in predicate calls in the body are distinct variables, explicit unifications are generated for these variables in the body goal, and complex unifications are broken down into several simpler ones. A goal ($Goal$) is either an atom or a number of goals connected by *conjunction*, *disjunction*, *if then else* or *not*. An atom ($Atom$) is either a procedure call ($PCall$) or a unification ($Unif$). A unification is either of the form $X = Y$ or $X = f(Y_1, \ldots, Y_n)$ where $X, Y, Y_1, \ldots, Y_n$ are variables ($\mathcal{V}ar$) and f a functor. Consider for example the definition of *append* in superhomogeneous form: [3]

$$\begin{aligned} append(X, Y, Z) \text{:-} \quad & X \Rightarrow [], Z := Y; \\ & X \Rightarrow [E \mid E_s], append(E_s, Y, R), Z \Leftarrow [E|R]. \end{aligned}$$

For a module M, $Proc_M$ denotes M's set of procedures. Every subgoal in such a procedure is uniquely identified by a *program point*. Put differently, a program point is associated with every atom and not-, conjunction-, disjunction- and if-then-else symbol. The set of all such program points is denoted by PP.

For analysis purposes, we consider for a goal G, its associated set of *execution paths*, $EP(G)$. An execution path is a sequence of program points identifying the atoms in G that can be encountered during a non-failing evaluation of G. Sequences are denoted by $\langle a_1, \ldots, a_n \rangle$ and for two sequences e_1 and e_2, $e_1 \bullet e_2$ denotes their concatenation. For a goal G, $EP(G)$ can be formally defined by:

Definition 2.3 $EP(A) = \langle pp \rangle$ for $A \in Atom$ identified by $pp \in PP$. For $G, G_1, G_2, G_3 \in Goal$, $EP(G_1, G_2) = \bigcup e_1 \bullet e_2$ for all $e_1 \in EP(G_1)$ and $e_2 \in EP(G_2)$. $EP(not\ G) = EP(G)$, $EP(G_1; G_2) = EP(G_1) \cup EP(G_2)$, and $EP(if\ G_1\ then\ G_2\ else\ G_3) = \bigcup(e_1 \bullet e_2) \cup \bigcup(e_1 \bullet e_3)$ for all $e_1 \in EP(G_1), e_2 \in EP(G_2)$ and $e_3 \in EP(G_3)$.

The *append* procedure given before contains two execution paths: one for each branch of the disjunction. We use $\{1\}$ to denote the first branch, $\{2\}$ to denote the second one.

We consider procedures that are mode correct when evaluated from left to right [2]. In order to check mode correctness, the compiler performs a *mode analysis*: For each procedure, the data flow in the procedure is recorded: starting from the initial instantiatedness graphs of the procedure's arguments, the analysis determines the exact atoms(s) where each node of a variable's

[3] In Mercury, unifications are *directed* (the direction being given by the mode function $\mathcal{I}$ - see further) and can be expressed as *test, assignment, construction,* and *deconstruction.* In this paper, however, we made this information explicit by using $\Rightarrow$, $\Leftarrow$, == and := for respectively deconstruction, construction, test and assignment.

value (corresponding to a type node in the variable's type graph) gets bound to a functor. An important observation is that each such node gets bound in maximally one atom in every execution path of the procedure's body goal. The result of mode analysis can be described by a function $\mathcal{I} : Proc_M \times PP \times \mathcal{V}ar \mapsto 2^{TPath}$ where $\mathcal{I}(p, pp, V) = \{\delta_1, \ldots, \delta_n\}$ denotes the set of nodes of V's value that get bound to a functor in the atom identified by pp in p's body.

2.2 Call-independent Data flow Analysis

The instantiatedness graphs used by the mode analysis, denote the instantiatedness of their associated variables *at run-time*, when the program is executed with complete input. During binding-time analysis, we are interested in another kind of instantiatedness graphs: namely instantiatedness graphs that describe instantiatedness of their associated variables *at specialisation-time*, i.e. when the program is run with *incomplete* input. We will refer to such an instantiatedness graph as the *binding-time* of a variable. A *call pattern* for a procedure p/n is a sequence of n binding-times, that describe the instantiatedness of p's arguments at procedure entry *during specialisation*. The set of all call patterns is denoted by $Callp$.

Example 2.4 Reconsider Figure 1, where $\tau_{il_1}, \tau_{il_2}, \tau_{il_3}$ now denote possible binding-times for a variable of type $list(T)$ and the definition of *append* (Example 2.2). Although *append*'s mode declaration says that the first two arguments of *append* should be completely ground at procedure call, a possible call pattern for *append* is $\langle \tau_{il_2}, \tau_{il_2}, free \rangle$ denoting a call to append *during specialisation* with the first two arguments only partially known, here bound to a list skeleton.

Instead of computing a single binding-time for each program variable (corresponding to a monovariant binding-time analysis), we compute different binding-times for a variable X in a procedure p, depending on:

- the call pattern p is called with. This comprises a polyvariant BTA: a number of different binding-times for the same variable inside a procedure p are computed, one for each call pattern the procedure is called with during analysis.
- the execution path in p: instead of associating a binding-time to a variable X, we associate a binding-time to an occurrence of X on an execution path in p, allowing X to have a different binding-time on different execution paths in p.

The set of binding-times associated to a variable X defined in a procedure p can thus be represented by a function $\mathbf{X} : EP(G) \times TPath \times Callp \mapsto \mathcal{B}$. If $\langle \eta_1, \ldots, \eta_n \rangle$ denotes a call pattern for p and δ is a node in X's type graph, then for an execution path $e \in EP(G)$, $\mathbf{X}(e, \delta, \langle \eta_1, \ldots, \eta_n \rangle) = bound$ denotes that the node δ of X gets bound on execution path e in G when p is called with $\langle \eta_1, \ldots, \eta_n \rangle$. Note that the exact atom in which δ is bound is given by

the mode function $\mathcal{I}$.

In order to be useful for program specialisation, the computed binding-times should form a *congruent division* [1]: for our BTA, this means that any node in a computed binding-time that depends (through data flow) on another node that is characterised as $free$, should itself be characterised as $free$. To be as precise as possible, the computed binding-times should incorporate as much static nodes as possible, without violating the congruence requirement. Binding-times are thus computed by examining the data flow inside a procedure, starting from the input nodes from the procedure's call pattern. Instead of performing a data flow analysis pure-sang over the domain of binding-times (requiring a reanalysis of the same procedure for every new encountered call pattern, as in [3]), we divide the BTA in three phases:

(i) Each procedure p/n is analysed only once: during this analysis, the data flow between the variables of p is made explicit, resulting in a number of symbolic constraints on the binding-times to be created, ensuring that the congruence requirement is satisfied. The binding-times participating in these constraints are regarded as parametrised w.r.t. p's call pattern. Consider for example the deconstruction $X \Rightarrow [E|E_s]$ on execution path $\{2\}$ in the *append* example from before. Since the data flows from X into E and E_s, the congruence requirement requires the following constraints on possible binding-times for E:

$$\mathbf{E}(\{2\}, \langle\rangle) \geq \mathbf{X}(\{2\}, \langle([|], 1)\rangle)$$
$$\mathbf{E}_s(\{2\}, \langle\rangle) \geq \mathbf{X}(\{2\}, \langle\rangle)$$
$$\mathbf{E}_s(\{2\}, \langle([|], 1)\rangle) \geq \mathbf{X}(\{2\}, \langle([|], 1)\rangle)$$

The first of these constraints, for example, states that under *any* call pattern, the label of the only node of E's binding-time (i.e. the root node $\langle\rangle$) on execution path e should be at least as dynamic as the node identified by $\langle([|], 1)\rangle$ in X's binding-time on e.

(ii) The constraints created in (i) typically include a lot of *local* dependencies, that can be resolved while remaining parametric w.r.t. p's call pattern. This phase is called *constraint reduction*. Consider, for example, the following two constraints (the first one generated from the atom $Z \Leftarrow [E|R]$ in the same *append* example, the second one is taken from (i)):

$$\mathbf{Z}(\{2\}, \langle([|], 1)\rangle) \geq \mathbf{E}(\{2\}, \langle\rangle)$$
$$\mathbf{E}(\{2\}, \langle\rangle) \geq \mathbf{X}(\{2\}, \langle([|], 1)\rangle)$$

The first of these constraints can be reduced (by unfolding it w.r.t. the second) to

$$\mathbf{Z}(\{2\}, \langle([|], 1)\rangle) \geq \mathbf{X}(\{2\}, \langle([|], 1)\rangle)$$

The binding-time nodes on both right-hand sides now denote input nodes from the call pattern of the *append* procedure (since all nodes of X's type

graph are input to *append*), meaning that they cannot be further reduced, since no actual call pattern is known.

(iii) Given a call pattern for a procedure p, the constraints associated to p can be evaluated w.r.t. this call pattern. Since after constraint reduction, these constraints do not contain local dependencies between them, evaluation can be performed quit efficiently, resulting in concrete binding-times for the involved variables.

In order to give a formal definition of the constraints generated for the atoms in a procedure, we introduce the following short-hand notation: For a set of execution paths $\mathcal{S}$, $\mathbf{X}^{\delta}_{\mathcal{S}}(\eta_1, \ldots, \eta_n)$ denotes $\bigcup_{e \in \mathcal{S}} \mathbf{X}(e, \delta, \langle \eta_1, \ldots, \eta_n \rangle)$

Definition 2.5 For a procedure $p(F_1, \ldots, F_n) : -G$ in $Proc_M$, we define its *associated set of constraints*, $\mathcal{C}_p = \bigcup \mathcal{C}_A$, for each atom A of G, where $\mathcal{C}_A$ is created in the following way: let $\mathcal{S} \subseteq EP(G)$ be the set of execution paths containing the atom A. If A is of the form:

- $X = Y$, $\mathcal{C}_A = \begin{array}{l} \{\mathbf{Y}^{\delta}_{\mathcal{S}}(F_1, \ldots, F_n) \geq \mathbf{X}^{\delta}_{\mathcal{S}}(F_1, \ldots, F_n) \mid \forall \delta \in \mathcal{I}(p, A, Y)\} \cup \\ \{\mathbf{X}^{\delta}_{\mathcal{S}}(F_1, \ldots, F_n) \geq \mathbf{Y}^{\delta}_{\mathcal{S}}(F_1, \ldots, F_n) \mid \forall \delta \in \mathcal{I}(p, A, X)\} \end{array}$
- $X = f(Y_1, \ldots, Y_n)$,
  $\mathcal{C}_A = \begin{array}{l} \bigcup_i \{\mathbf{Y}_{\mathbf{i}\mathcal{S}}^{\delta}(F_1, \ldots, F_n) \geq \mathbf{X}^{\langle f/i \rangle \bullet \delta}_{\mathcal{S}}(F_1, \ldots, F_n) \mid \forall \delta \in \mathcal{I}(p, A, Y_i)\} \cup \\ \bigcup_i \{\mathbf{X}^{\langle f/i \rangle \bullet \delta}_{\mathcal{S}}(F_1, \ldots, F_n) \geq \mathbf{Y}_{\mathbf{i}\mathcal{S}}^{\delta}(F_1, \ldots, F_n) \mid \forall \langle f/i \rangle \bullet \delta \in \mathcal{I}(p, A, X)\} \end{array}$
- $q(X_1, \ldots, X_k)$ with $q(L_1, \ldots, L_k) : -G_q \in Proc_M$:
  $\bigcup_i \{\mathbf{X}_{\mathbf{i}\mathcal{S}}^{\delta}(F_1, \ldots, F_n) \geq \mathbf{L}_{\mathbf{i}EP(G_q)}^{\delta}(\mathbf{X}_{\mathbf{1}\mathcal{S}}(F_1, \ldots, F_n), \ldots, \mathbf{X}_{\mathbf{k}\mathcal{S}}(F_1, \ldots, F_n)) \mid$
  $\forall \delta \in \mathcal{I}(p, A, X_i)\}$

In case of a unification, the created constraints express that, if there is data flow for a node δ of X to a node γ of Y, then if δ is *free* then the binding-time of node γ should also be *free*. In case of a procedure call to q with actual arguments $X_1, \ldots X_k$ (being variables of p), then the binding-times for the *output* nodes δ of these variables should be *free* if the corresponding nodes of q's formal arguments are *free* when these are evaluated with a call pattern consisting of the binding-times of $X_1, \ldots X_n$.

Example 2.6 For the *append* procedure, the set $\mathcal{C}_{append}$ is as follows:

$\mathbf{Z}^{\langle\rangle}_{\{1\}}(X, Y, Z) \geq \mathbf{Y}^{\langle\rangle}_{\{1\}}(X, Y, Z)$
$\mathbf{Z}^{\langle./1\rangle}_{\{1\}}(X, Y, Z) \geq \mathbf{Y}^{\langle./1\rangle}_{\{1\}}(X, Y, Z)$
$\mathbf{E}^{\langle\rangle}_{\{2\}}(X, Y, Z) \geq \mathbf{X}^{\langle./1\rangle}_{\{1\}}(X, Y, Z)$
$\mathbf{E}_{\mathbf{s}}{}^{\langle\rangle}_{\{2\}}(X, Y, Z) \geq \mathbf{X}^{\langle\rangle}_{\{1\}}(X, Y, Z)$
$\mathbf{R}^{\langle\rangle}_{\{2\}}(X, Y, Z) \geq \mathbf{Z}^{\langle\rangle}_{\{1,2\}}(\mathbf{X}_{\{2\}}(X, Y, Z), \mathbf{Y}_{\{2\}}(X, Y, Z), \mathbf{Z}_{\{2\}}(X, Y, Z))$
$\mathbf{R}^{\langle./1\rangle}_{\{2\}}(X, Y, Z) \geq \mathbf{Z}^{\langle./1\rangle}_{\{1,2\}}(\mathbf{X}_{\{2\}}(X, Y, Z), \mathbf{Y}_{\{2\}}(X, Y, Z), \mathbf{Z}_{\{2\}}(X, Y, Z))$
$\mathbf{Z}^{\langle./1\rangle}_{\{2\}}(X, Y, Z) \geq \mathbf{E}^{\langle\rangle}_{\{2\}}(X, Y, Z)$

$\mathbf{Z}^{\langle\rangle}_{\{2\}}(X,Y,Z) \geq \mathbf{R}^{\langle\rangle}_{\{2\}}(X,Y,Z)$

Given the binding-times for an initial procedure call, the result of BTA for a module M is the *least solution* (in the sense that as few nodes as possible have a value *free*) to the constraint system $\mathcal{C}_M$, where $\mathcal{C}_M = \cup\mathcal{C}_p$ for each $p \in Proc_M$. For a set of constraints $\mathcal{C}$, $\overline{\mathcal{C}}$ denotes the set consisting of the reduced constraints of $\mathcal{C}$. Of course, reduction is required to preserve the least solution.

Strategy
A number of additional constraints are created that link the binding-time of a variable's node to the reducibility at specialisation-time of the atom in which it is bound: when an atom is residualised, the nodes of its arguments that are bound by this atom should be made *free* – reflecting their state of instantiation at specialisation-time. Likewise, the decisions when to reduce or residualise an atom (depending on the binding-times of the nodes that are input to the atom and the latter's place in the control flow) can also be expressed by additional constraints. In the remainder of this paper, we assume $\mathcal{C}_p$ to include these constraints.

Binding-time Analysis of a Multi-module Program
Given a module M that does not import anything, $\overline{\mathcal{C}}_M$ can be computed straightforwardly. In case M imports from $M_1, \ldots, M_n$, $\overline{\mathcal{C}}_M$ can be computed given only $\overline{\mathcal{C}}_{M_1}, \ldots, \overline{\mathcal{C}}_{M_n}$: in order to normalise the constraints of M, the normalised constraints from $M_1, \ldots, M_n$ can be used without the need to reanalyse or renormalise the constraints of these modules. As such, the call-independent data flow analysis can be performed bottom-up in a modular way, without the need to reanalyse a module when it is imported in a module to be analysed.

2.3 From Binding-times to Annotations

One way of representing the results obtained by BTA consists of annotating the original source program by instructions, specifying what goals are to be reduced at specialisation-time and what goals should be residualised. Annotation is a call-dependent process: given a call to $p(F_1, \ldots, F_n) : -G$ with call pattern $\eta_1, \ldots, \eta_n$, $\overline{\mathcal{C}}_p$ is evaluated w.r.t. $\eta_1, \ldots, \eta_n$ to obtain binding-times for every variable in G. These binding-times are used to annotate each atom in G as either reducible or non-reducible. If during annotating G, a call to q with call pattern $\gamma_1, \ldots, \gamma_k$ is marked as reducible (denoting it should be unfolded during specialisation), an annotated version of q must be created w.r.t. $\gamma_1, \ldots, \gamma_n$.

Example 2.7 With the possible instantiatedness graphs for $list(T)$ as depicted in Figure 1, there are 9 possible call patterns for *append*. From these,

only 3 are really interesting: $\langle free, free, free\rangle$,$\langle \tau_{il_1}, \tau_{il_1}, free\rangle$ and $\langle \tau_{il_2}, \tau_{il_2}, free\rangle$. We show two possible annotated versions of *append*: for the call pattern $\langle \tau_{il_2}, \tau_{il_2}, free\rangle$, denoting that the two input arguments of *append* are bound to a list skeleton, and for $\langle free, free, free\rangle$, denoting that the two input arguments are unknown. In the annotated versions, underlined variables denote variables that are free at specialisation-time, not-underlined variables are bound (at least to an outermost functor) at specialisation time. Atoms annotated with the superscript s denote they can be evaluated at specialisation time:

- *append* with call pattern $\langle \tau_{il_2}, \tau_{il_2}, free\rangle$:

  $append(X, Y, Z)$:- $X \Rightarrow^s [], Z := Y;$
  $X \Rightarrow^s [\underline{E} \mid E_s], append(E_s, Y, R)^s, Z \Leftarrow [\underline{E}|R].$

- *append* with call pattern $\langle free, free, free\rangle$:

  $append(\underline{X}, \underline{Y}, \underline{Z})$:- $\underline{X} \Rightarrow [], \underline{Z} := \underline{Y};$
  $\underline{X} \Rightarrow [\underline{E} \mid \underline{E_s}], append(\underline{E_s}, \underline{Y}, \underline{R}), \underline{Z} \Leftarrow [\underline{E}|\underline{R}].$

In principle, also the annotation phase can be made modular: Since the (finite) type graphs and modes for all arguments of a procedure p are known, all possible call patterns for p can be enumerated. $\overline{\mathcal{C}}_p$ can be evaluated w.r.t. these call patterns and for those patterns for which a call to p can be made reducible, an annotated version of p can be created. For a procedure p, let $\mathcal{A}_p$ denote the set of all such annotated versions of p. If a call to p w.r.t $\eta_1, \ldots, \eta_n$ is encountered, $\overline{\mathcal{C}}_p$ is evaluated and if the call is annotated reducible, it is renamed to the right version in $\mathcal{A}_p$. For a multi module program $M_1, \ldots, M_n$, $(\overline{\mathcal{C}}_{M_1}, \mathcal{A}_{M_1}), \ldots, (\overline{\mathcal{C}}_{M_n}, \mathcal{A}_{M_n})$ can be computed in a modular, bottom-up fashion, where $\mathcal{A}_M = \bigcup \mathcal{A}_p \forall p \in Proc_M$.

In practice, a time-space tradeoff needs to be considered: for some predicates p, the set $\mathcal{A}_p$ may contain a considerable number of annotated versions (depending on the number of p's arguments and their possible binding-times) – some of which may never be needed. In that case, a combined approach may be feasible: a subset of $\mathcal{A}_p$ is generated in a call-independent way (for "frequently occurring" call patterns), which is gradually extended when calls occur for which no annotated version is yet available. In such situations, however, the definition of p must be available, sacrifying a purely modular annotation phase.

3 Discussion

We have discussed how both the computation of binding-times and the annotation of a Mercury source program can be achieved in a call-independent way. This enables BTA to be performed one module at a time, *and* efficiently, since the hard part of the dataflow analysis (constraint normalisation) needs only to be performed once for each module. As such, our analysis tries to overcome the problems with a call-dependent analysis sketched in the introduction.

The described BTA by constraint normalisation was implemented (at the moment for one module and for predicates having *in*/*out* modes only). Topics of ongoing research include extending the analysis to deal with higher-order constructs, extending the implementation to deal with modules and Mercury's full mode system, experimenting with different specialisation strategies and performing some benchmarks on small and larger programs.

Acknowledgments

The authors wish to thank Danny De Schreye and Karel De Vlaminck for their continuous interest in this work. They also wish to thank anonymous referees for valuable comments which helped to improve the current paper.

References

[1] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[2] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1–3):17–64, October–November 1996.

[3] W. Vanhoof and M. Bruynooghe. Binding-time analysis for mercury. In D. De Schreye, editor, *16th International Conference on Logic Programming*. MIT Press, 1999. To Appear.