Towards a Modular Binding-time Analysis for Higher-order Mercury ⋆

Wim Vanhoof and Maurice Bruynooghe

Department of Computer Science, K.U.Leuven, Belgium
{wimvh,maurice}@cs.kuleuven.ac.be

Abstract. This extended abstract describes work in progress on binding-time analysis (BTA) for Mercury. It reports on a so called *phased* BTA that deals with the higher-order aspects of Mercury, and of which a large part can be performed on a modular basis.

1 Introduction and Context

Mercury is a recently introduced logic programming language, comprising a lot of features needed for modern software engineering practice: polymorphism, type-classes and a strong module system are some examples of the means available to the programmer to design and build modular programs that employ a lot of abstraction and reuse of general components.

Employing abstraction and generality, however, imposes a penalty on the efficiency of the resulting program due to the presence of for example procedure calls and tests for which the input is (partially) known at compile-time. To overcome this performance problem, the Mercury compiler performs a lot of optimizations on the original source code. Although most of these optimizations are implemented as different processes during compilation, some of them are instances the more general framework of partial evaluation. Examples are inlining of procedure bodies, higher-order call specialisation, specialisation of type-information, etc. A problem shared by these optimizations is knowing at what points in the code enough information is present to apply the optimization under consideration. Currently, this problem is solved mostly by using some heuristics that are hard coded in the analysis.

A more general approach is the use of binding-time analysis (BTA) to perform a thorough data flow analysis, and propagate information through the program about what variables are definitely bound to a value at compile-time, independently of the program's runtime input. Such information is crucial if partial evaluation is to perform optimizations that are correct and beneficial. An advantage of the more general approach is that the results of BTA can be

⋆ Extended Abstracts of LOPSTR 2000, Tenth International Workshop on Logic-based Program Synthesis and Transformation, 24-28 July 2000, London, UK. Technical Report Report Series, Department of Computer Science, University of Manchester, ISSN 1361-6161. Report number UMCS-00-6-1. http://www.cs.man.ac.uk/cstechrep/titles00.html

shared by more than one optimization. As such, different optimizations do not need to redo the analysis, additional optimizations can more easily be plugged in, and precision improvements of the BTA have an impact on a broader scope of optimizations. The results of BTA can be shown as annotations on the original source code and provide as such excellent feedback to the user, enabling a better understanding of why several optimizations are (not) performed.

Binding-time analysis has been studied thoroughly before, namely in the context of functional (e.g. [2, 4]) and imperative (e.g. [1]) languages. In a logic programming setting, however, little attention has been paid to BTA [6, 7, 3]. In previous work [10] we have defined a completely automatic BTA for a subset of the logic programming language Mercury that does not require any help from the user. The current work combines research on different topics in BTA in different paradigms and extends them to deal with a logic programming language, in casu Mercury. It reformulates and extends our previous work extensively: in contrast with [10], our BTA now deals with the higher-order aspects of Mercury, and allows to be performed on the same modular basis as compilation which is crucial if the analysis has to be applicable for real, multi-module applications. Modular analysis recently gained attention in a logic programming setting (e.g. [8]). The basic ideas for a modular approach to BTA for a first-order subset of Mercury were explored in [11].

2 A domain for precise BTA

Binding-time analysis is about knowing *what* runtime values will be known already at compile-time, without being interested in the concrete values. In [10], we propose a domain of binding-times for first-order values that can deal with *partially instantiated* data structures: data structures of which some parts are known at compile-time, whereas others are not (e.g. [5]).

In a statically typed language like Mercury, the set of possible values a program variable can have at runtime can be described at compile time by a finite type graph. A binding-time describing a first-order value is obtained by labelling each or-node in the value's type graph as either *static* or *dynamic* meaning that the corresponding subterm is either known or not at compile-time. Consider for example the definition of a type `list(T)`, denoting a polymorphic list, where such a list is either the empty list (denoted by []) or a cons (denoted by [|]) of a value of type T and a list of T. Figure 1 shows a possible type graph for this type, with the possible corresponding binding-times. In the figure, τ_1, τ_2 and τ_3 represent respectively a value that is completely known at compile-time, a value for which the skeleton of the list is definitely known at compile-time but the elements are not, and a value that is possibly completely unknown at compile-time. To obtain a suitable domain of binding-times, we impose the restriction that all descendants of a *dynamic* node must also be *dynamic*.

Mercury is a higher-order language, meaning that predicates can be passed around as arguments of other predicate calls. Though Mercury also allows the construction of *closures*, we do not consider these in this extended abstract.

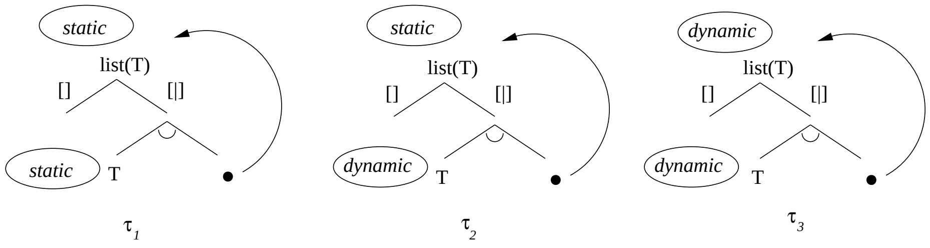

Fig. 1. Labelled type graphs for `list(T)`.

A Mercury program (transformed to superhomogeneous form [9]) consists of a number of procedures, which are predicates with precisely one associated mode declaration. The body of a procedure is represented by a single clause in which every atom is either a unification or a call. The arguments of atoms are distinct variables and complex unifications are broken into simpler ones. Since Mercury procedures must be well-moded, the direction of every unification is known at compile-time. In the examples, we use $\Rightarrow$, $\Leftarrow$, $==$ and $:=$ to denote deconstruction, construction, test and assignment.

The binding-time of a local variable is determined by the data flow inside the procedure. The basic problem when analysing a procedure involving higher-order calls, is that the data flow between such a call's arguments is determined by the value of the higher-order variable. Consequently, without knowing (an approximation of) these values, it is impossible to compute precise binding-times for such a call's output arguments. Consider the following example:

Example 1. The *map* procedure converts the first list of type T to a new list of type T using the predicate provided in the second argument.

$$
\begin{array}{l}
:-pred\ map(list(T), pred(T,T), list(T)).\\
:-mode\ map(in, in(pred(in, out)), out)\ is\ det.\\
map(L_1, P, L_0) :- L_1 \Rightarrow [], L_0 \Leftarrow [];\\
\qquad\qquad\qquad\quad L_1 \Rightarrow [E_1|Es_1], P(E_1,E_2), map(Es_1, P, Es_2), L_0 \Leftarrow [E_2|Es_2].
\end{array}
$$

In Example 1, we know from the mode declaration that E_1 is input and E_2 output from the call to P. However, to compute a precise binding-time for E_2, we need to consider the data flow between E_1 and E_2 which is, in turn, determined by the concrete procedure P will be bound to at runtime. Traditionally, the set of procedures a higher-order variable can be bound to at run-time is computed by a *closure analysis*, that is run before the actual BTA [5]. In our current approach, it turns out to be convenient to use a single domain for BTA and closure analysis.

To redefine the notion of a binding-time in such a higher-order context, we introduce the domain $\mathcal{B} = \{static, dynamic\} \cup \{static(S) \mid S \in 2^{Proc}\}$ on which the ordering defined by the lattice in Figure 2 is imposed. The ordering between elements $static(S_1)$ and $static(S_2)$ is determined by the ordering $\supseteq$ on the sets of procedures: $static(S_1) \geq static(S_2)$ if and only if $S_1 \supseteq S_2$. A *binding-time* then is defined as a labelling of type graph w.r.t. $\mathcal{B}$ such that the nodes representing

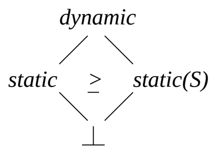

Fig. 2. The ordering on $\mathcal{B}$.

a first-order subvalue are labelled with *static* or *dynamic*, but the higher-order nodes are labelled with *static*(S) (S being a set of procedures) or *dynamic*. The domain of binding-times is denoted by $\mathcal{IT}$. Note that the ordering between elements of $\mathcal{B}$ induces an ordering between the elements of $\mathcal{IT}$. For $\tau_X, \tau_Y \in \mathcal{IT}$, we say that τ_X is *at least as dynamic* as τ_Y iff $\tau_X \geq \tau_Y$.

A node in a type graph is identified by the path (a sequence of functor/argument pairs) from the root towards it. The set of all such paths is denoted by *TPath*. Two paths are considered equivalent when they denote the same node, like e.g. $\langle\rangle$ (the root node) and $\langle([|], 2)\rangle$ in Figure 1. Given a value of a certain type and $\delta \in \mathit{TPath}$, τ^{δ} denotes the subvalue of τ originating in the node δ.

3 Modular BTA for Mercury

A Mercury program consists of a number of type declarations and procedure definitions that are generally distributed over a number of different modules $M_1, \ldots, M_n$. Each such module exports a number of its definitions (we call these *interface* declarations), whereas others are for internal use only.

In [10] we formulated BTA for a subset of Mercury as a top-down, call dependent analysis. Extending such a call dependent analysis to deal with multi-module programs is nontrivial, the main issues being that 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, and the called predicates of a module M are re-analysed in every program M is used in.

To overcome these problems, we split the process of BTA in 4 phases, depicted in Figure 3, of which the first two are performed independent of the initial call and call pattern of interest. Hence, the result of these two phases can be stored and reused whenever a program is analysed that imports the module M. The two subsequent phases do depend on the call and call pattern of interest, and hence need to be redone every time a procedure needs to be analysed w.r.t. a call pattern not encountered before. In what follows we will give the main intuitions behind each phase.

3.1 Phase-1: Collecting data dependencies

Instead of starting from a call pattern for a procedure p and propagating the binding-times over p's body atoms, the first part of our analysis does analyse p's

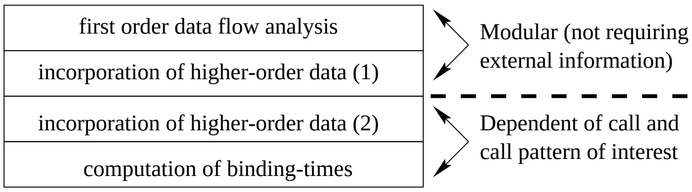

Fig. 3. Phased BTA.

body atoms, but generates a set of *data dependencies* that reflect the data flow inside p. Consider the following example:

Example 2. Consider $append(list(T) :: in, list(T) :: in, list(T) :: out)$

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

Translating a unification in superhomogeneous form to a set of data dependencies is straightforward: If we use X^δ to denote the subvalue of X originating in the node $\delta \in TPath$, then we can express the data flow inside the unification $X \Rightarrow [E \mid E_s]$ by $E \leftarrow X^{\langle([|],1)\rangle}$ and $E_s^{\langle\rangle} \leftarrow X^{\langle\rangle}$ (due to the equivalence between $\langle\rangle$ and $\langle([|],2)\rangle$). In what follows, we simply write X instead of $X^{\langle\rangle}$. Likewise, the unification $Z \Leftarrow [E|R]$ results in the dependencies $Z^{\langle([|],1)\rangle} \leftarrow E$ and $Z \leftarrow R$. The intended meaning of a dependency like $E_s \leftarrow X$ is that whatever the binding-time of X is (say τ_X), the binding-time of E_s (denoted by τ_{E_s}) should be at least as dynamic as the binding-time of X: $\tau_{E_s} \geq \tau_X$. For completeness sake, we mention that the right hand side of such a dependency can also be one of the constants *static*, *dynamic*, or *static*(S). The set of all such dependencies is denoted with $\mathcal{D}_\mathcal{D}$.

Due to the well-modedness of M's procedures, it is possible to trace the data flow back to the procedure's input arguments, and consequently to express the data dependencies on a variable in function of the procedure's (input) arguments. Data dependencies in this format are said to be in *normal form*, and we call the set of data dependencies in normal form on a procedure p's (output) arguments p's normal form. In the append example, the dependencies $E \leftarrow X^{\langle([|],1)\rangle}$ and $E_s \leftarrow X$ are in first order normal form, whereas $Z^{\langle([|],1)\rangle} \leftarrow E$ and $Z \leftarrow R$ are not.

A call introduces data flow dependencies on its output arguments, expressed in function of its input argument, by renaming the called procedure's normal form w.r.t. the actual variables from the call.

Example 3. Reconsider Example 2. Suppose $Z \leftarrow Y$ was already derived. The call $append(E_s, Y, R)$ introduces the renamed dependency $R \leftarrow Y$.

Ultimately, we want every dependency to be in first order normal form. For a dependency $X^\delta \leftarrow Y^\gamma$, this can be achieved by unfolding Y^γ w.r.t. all dependencies on (subvalues of) Y that share a subvalue with Y^γ. We say that two subvalues of a variable, X^{δ_1} and X^{δ_2} *share a subvalue* if and only if δ_1 is an extension of δ_2 or vice versa. More formally, if we denote with $\langle\rangle$ the empty sequence and with $\alpha \bullet \beta$ the concatenation of α with β, we define the following function:

Definition 1. *We define* $ext : TPath \times TPath \mapsto TPath \times TPath$ *as follows:*

$$\begin{array}{l} ext(\gamma, \delta) = (\langle\rangle, \eta) \text{ if } \gamma = \delta \bullet \eta \\ ext(\gamma, \delta) = (\eta, \langle\rangle) \text{ if } \gamma \bullet \eta = \delta \end{array}$$

and undefined otherwise. Note that if $ext(\gamma, \delta) = (\alpha, \beta)$ *then* $\gamma \bullet \alpha = \delta \bullet \beta$. *Unfolding a dependency* $X^\delta \leftarrow Y^\gamma$ *w.r.t. a set of dependencies* $\mathcal{D}$ *is defined as*

$$unf(X^\delta \leftarrow Y^\gamma, \mathcal{D}) = \{X^{\delta\bullet\alpha} \leftarrow Z^{\eta\bullet\beta} \mid Y^{\gamma'} \leftarrow Z^\eta \in \mathcal{D} \text{ and } ext(\gamma, \gamma') = (\alpha, \beta)\}$$

Example 4. Unfolding $Z^{\langle([|],1)\rangle} \leftarrow E$ w.r.t. $E \leftarrow X^{\langle([|],1)\rangle}$ results in $\{Z^{\langle([|],1)\rangle} \leftarrow X^{\langle([|],1)\rangle}\}$. Unfolding $Z \leftarrow R$ w.r.t. $R^{\langle([|],1)\rangle} \leftarrow E_s^{\langle([|],1)\rangle}$ results in $\{Z^{\langle([|],1)\rangle} \leftarrow E_s^{\langle([|],1)\rangle}\}$.

For a set of dependencies $\mathcal{D}$, let *compact*($\mathcal{D}$) denote the process of repeatedly selecting a dependency from $\mathcal{D}$ that is not in first order normal form and unfolding it w.r.t. $\mathcal{D}$, until a fixed point is reached.

The first phase of analysing a module M then consists of computing a table μ_M, mapping every procedure to its normal form. We suppose that the imported modules have been analysed before, and hence the normal form of procedures called but not defined in M are available and can be entered in the table. The table is then completed by repeatedly combining, for each procedure p from M, the dependencies from the atoms in p (using the table for retrieving the normal form of the calls) and use *compact* to compute a new normal form for p, replacing an earlier entry for p in the table. It can be proven that a fixed point is reached after a finite number of steps, and the entries in the table are all in normal form.[1] The result of analysing *append* (see Example 2) is as follows:

$$\begin{array}{lll} Z \leftarrow Y & E \leftarrow X^{\langle([\,|\,],1)\rangle} & R \leftarrow Y \\ Z^{\langle([\,|\,],1)\rangle} \leftarrow X^{\langle([\,|\,],1)\rangle} & Es \leftarrow X & R^{\langle([\,|\,],1)\rangle} \leftarrow X^{\langle([\,|\,],1)\rangle} \end{array}$$

3.2 Phase 2: (Partly) incorporating the higher-order data flow

For procedures that do not involve any higher-order calls, their computed normal form is *complete*: if there is a data flow path in the procedure from (a subvalue of) an input variable to (a subvalue of) an output variable, it is represented by a dependency in first order normal form. When a procedure does involve a higher-order call, its computed normal form is not complete, since the dependencies between the call's arguments can not be incorporated without knowing the possible procedures the higher-order variable can be bound to (Reconsider Example 1). Like in that example, higher-order variables are often part of the

[1] Note that the table can be build efficiently by splitting the module's call graph into strongly connected components and compute, in a bottom-up fashion, a fixed point for every strongly connected component.

procedure's argument list. Hence, during phase 2, we convert μ_M to another table that associates a set of dependencies to a combination of a procedure and a call pattern. This enables to associate different sets of dependencies to a procedure, depending on the call pattern.

Let us first show how to compute the binding-time of a variable X in a predicate, given a set of data dependencies in first order normal form on that variable and a call pattern for the predicate. We define a call pattern (*Callp*) as a sequence of binding-times for the predicate's arguments. For every data dependency $X^\delta \leftarrow \ldots$, the right hand side of the dependency is evaluated resulting in a binding-time τ. The final binding-time of X (say τ_X) is defined as the *least dynamic* binding-time such that $\tau_X^\delta \geq \tau$. Evaluating the right hand side of a dependency in first order normal form is easy: either it is a constant (*static*, *dynamic*, or *static*(S)), or else it is a (subvalue of) an input argument of the procedure, and hence its binding-time can be retrieved from the call pattern. To compute binding-times in this way, we introduce the function $\beta : \mathit{Proc} \times \mathit{Callp} \times 2^{\mathcal{D}_\mathcal{D}} \times \mathcal{V}ar \mapsto \mathcal{IT}$.

The new table, which we represent by a function $\nu : \mathit{Proc} \times \mathit{Callp} \mapsto 2^{\mathcal{D}_\mathcal{D}}$ is built by a call dependent process. The first entry that is constructed is $\nu(p, \pi)$, for an initial predicate call to p w.r.t. call pattern π. If, during construction, the dependencies $\nu(q, \pi')$ are retrieved while there is not yet an entry for (q, π') in ν, such an entry is added with an initial value of $\mu_n(q)$ (the result of phase 1). Entries are reconstructed until a fixed point on ν has been reached. Rather than sketching the complete process, we emphasize on the handling of predicate calls when constructing an entry for p w.r.t. call pattern π. For a first-order call $q(V_1, \ldots, V_n)$, the binding-times of $V_1, \ldots, V_n$ are computed using $\beta(p, \pi, \nu(p, \pi), V_i)$. Together, they constitute a call pattern π' that specifies what particular set of dependencies of q should be retrieved: $\nu(q, \pi')$. For a higher-order call $X(V_1, \ldots, V_n)$, first the binding-time of X is computed and if $\beta(p, \pi, \nu(p, \pi), X) = \mathit{static}(Q)$, then $\forall q \in Q$, we add dependencies for $q(V_1, \ldots, V_n)$ as in the first-order case. If $\beta(p, \pi, \nu(p, \pi), X) = \mathit{dynamic}$, we add $V_i \leftarrow \mathit{dynamic}$ for each output argument of the call V_i.

Like μ_n, the table ν still contains data dependencies instead of concrete binding-times. Since these dependencies are independent of the first-order parts of the call pattern, it is unnecessary to create different entries for (p, π) and (p, π') if the higher-order parts of π and π' are equal, since then $\nu(p, \pi) = \nu(p, \pi')$.

Taking only the higher-order parts of a call pattern into account when constructing ν has another advantage: if a procedure calls another procedure with a higher-order parameter, often the higher-order part of the callee's call pattern is independent of the caller's call pattern. Consider the following code fragment, illustrating a typical use of a higher-order procedure.

$p(list(int) :: in, list(int) :: out).$ $\quad$ $plus1(int :: in, int :: out).$
$p(L_1, L_2) : -P \Leftarrow plus1, map(P, L_1, L_2)$ $\quad$ $plus1(X, Y) : -Y = X + 1.$

Since $P \leftarrow \mathit{static}(plus1)$ will be part of the dependencies generated for p, using β to evaluate P will result in $\mathit{static}(plus1)$. Hence a particular entry in ν is created for *map* w.r.t. $P = plus1$. A complete set of data dependencies can be constructed for *map* w.r.t. this particular call pattern, again resulting in a complete set of data dependencies for p *independent* of p's call pattern.

This example shows why we split the computation of ν into two separate phases. In the first one, we do not consider any particular calls to the interface predicates of the module from outside the module, and hence also this part can be run once and for all for the module. This is achieved by constructing the table for the initial entries $(p_i, \langle dynamic, \ldots, dynamic\rangle)$ for all interface predicates p_i. The resulting table is denoted with ν_0. Using $\langle dynamic, \ldots, dynamic\rangle$ as call pattern makes sure that no information from outside the module is assumed, but all higher-order information that is internally present (like $P \leftarrow static(plus1)$ in the example above) will nevertheless be propagated, resulting in *complete* sets of data dependencies for those predicates in which the data flow does not rely on higher-order values in their arguments.

3.3 The final phases: computing binding-times

The final phases take a particular call to an interface predicate p_i into account. Let π_i denote the call pattern of interest. First, the table ν_0 must be completed with the dependencies introduced by the higher-order parts in π_i (if any). Therefore, the algorithm from the previous phase is rerun on ν_0 for the call (p_i, π_i). This constitutes phase 3 of the BTA; we denote the resulting table with ν_1. For each procedure/call pattern pair that has an entry in ν_1, the associated set of dependencies now is complete and in first order normal form.

It is only during the last phase (phase 4) that the first order parts of the call pattern are taken into account. Computing binding-times for the variables of p_i under call pattern π_i is straightforward: simply select the appropriate dependencies from ν_1 and evaluate them using β.

While the last phase does not involve any fixed point computation, in general one will be interested not only in computing the binding-times for a single procedure (given a call pattern), but also in computing the binding-times for the procedures that in turn are called from the former. This does require a fixed point computation, in which each round consists of selecting the right set of dependencies from ν_1 and evaluate them using β.

4 Final Remarks

We have described BTA as a phased process, of which a large part (phases 1 and 2) is performed without information about a particular call to an interface predicate. Hence, it suffices to run these phases once and for all for each module separately, bottom up in the module hierarchy. Phases 3 and 4 need to be rerun for every call of interest to an interface predicate: the former because the call pattern can include information necessary to complete the data flow analysis, the latter because in general binding-times for local- and output variables can only be computed in the presence of binding-times for the input arguments.

To perform some large scale experiments, we are strongly collaborating with the Mercury developers in Melbourne to implement and integrate the BTA into a version of the Mercury compiler. Some topics of ongoing work are the integration of some involved features like the use of typeclasses, abstract types and the presence of circular dependencies in the module structure of a Mercury program.

We thank Harald Søndergaard for fruitful discussions on the relation between groundness analysis and binding-time analysis. We also thank Zoltan Somogyi for providing the opportunity of working together with the Mercury team. Special thanks go to Tyson Dowd, Fergus Henderson, David Jeffrey, David Overton, Peter Ross, Mark Brown and Anthony Senyard for explaining over and over again several implementation issues of the Melbourne Mercury compiler, but perhaps most of all for providing an environment which made the first author's visit to Australia an unforgettable experience. We also thank anonymous referees for their helpful remarks. Wim Vanhoof is supported by the IWT, Belgium.

References

1. L. O. Andersen. Binding-time analysis and the taming of C pointers. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation. PEPM'93*, pages 47–58, New York, NY, USA, 1993. ACM Press.
2. A. Bondorf and J. Jørgensen. Efficient analyses for realistic off-line partial evaluation. *Journal of Functional Programming*, 3(3):315–346, July 93.
3. Maurice Bruynooghe, Michael Leuschel, and Kostis Sagonas. A polyvariant binding-time analysis for off-line partial deduction. In C. Hankin, editor, *Programming Languages and Systems, Proc. of ESOP'98, part of ETAPS'98*, pages 27–41, Lisbon, Portugal, 1998. Springer-Verlag. LNCS 1381.
4. F. Henglein and C. Mossin. Polymorphic binding-time analysis. *Lecture Notes in Computer Science*, 788, 1994.
5. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
6. J. Jørgensen and M. Leuschel. Efficiently generating efficient generating extensions in Prolog. In O. Danvy, R. Glück, and P. Thiemann, editors, *Proceedings Dagstuhl Seminar on Partial Evaluation*, pages 238–262, Schloss Dagstuhl, Germany, 1996. Springer-Verlag, LNCS 1110.
7. T. Mogensen and A. Bondorf. Logimix: A self-applicable partial evaluator for Prolog. In K.-K. Lau and T. Clement, editors, *Proceedings LOPSTR'92*, pages 214–227. Springer-Verlag, Workshops in Computing Series, 1993.
8. G. Puebla and M. Hermenegildo. Some issues in analysis and specialization of modular Ciao-Prolog programs. Las Cruces, 1999. Proceedings of the Workshop on Optimization and Implementation of Declarative Languages. In Electronic Notes in Theoretical Computer Science, Volume 30 Issue No.2, Elsevier Science.
9. 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.
10. W. Vanhoof and M. Bruynooghe. Binding-time analysis for Mercury. In D. De Schreye, editor, *16th International Conference on Logic Programming*, pages 500 – 514. MIT Press, 1999.
11. W. Vanhoof and M. Bruynooghe. Towards modular binding-time analysis for first-order Mercury. Las Cruces, 1999. Proceedings of the Workshop on Optimization and Implementation of Declarative Languages. In Electronic Notes in Theoretical Computer Science, Volume 30 Issue No.2, Elsevier Science.