Binding-Time Analysis for Mercury

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

Abstract

In this paper, we describe a binding-time analysis (BTA) for a statically typed and strongly moded pure logic programming language, in casu Mercury. Binding-time analysis is the key concept in achieving off-line program specialisation: the analysis starts from a description of the program's input available for specialisation, and propagates this information throughout the program, deriving directives for when and how to perform specialisation. Exploiting the possibilities offered by Mercury's strong type and mode system, we present a completely automatic BTA dealing with partially static binding-times.

1 Introduction

Program specialisation is a well-studied source-to-source transformation, capable of optimising a program P with respect to a known part s of its input (s, d): Some (ideally all) program parts depending only on s are evaluated at specialisation-time, whereas the remaining operations are residualised in a new program P_s. In general, $P_s(d)$ will compute more efficiently than $P(s, d)$ since the former has less operations to perform.

In the *off-line* approach to specialisation, the program is analysed using a description of the input available for specialisation (e.g. [7, 2]). Such a description states, in its most rudimentary form, which of the program query's input arguments are bound to a value at specialisation-time. The task of *binding-time analysis* (BTA) is to propagate this information from the query throughout the whole program, deriving *binding times* for every variable in the program, stating whether or not it is bound to a value at a specific program point.

From the result of this BTA, a set of directives is produced, specifying for each operation whether it can be performed at specialisation-time (when concrete input will be available), or should be residualised. Actual specialisation is achieved afterwards, by merely following the generated directives, running parts of the program on the available concrete input. However, all control decisions for the specialiser are taken beforehand, during BTA.

In a logic programming setting, most attention has been paid to *on-line* specialisation: Without prior analysis, the program is run with the available input under the supervision of a control system that decides –

during specialisation – what operations to evaluate or to residualise while guaranteeing termination. In general, an on-line system can achieve more thorough specialisation results due to the availability of concrete input to the control system [10, 11].

However, the off-line approach offers a lot of advantages: While the division in two phases makes the process conceptually cleaner, the output of BTA is interesting in its own right: The results of BTA can be used by other analyses apart from specialisation, or may be inspected by the user – enabling a better understanding of the process and the obtained results. Moreover, off-line techniques are generally considered to be more efficient than their on-line counterparts, and seem to be better suited for self-application [7].

Despite the advantages offered, little work on BTA for logic programming languages exists [13, 8, 3]. Moreover, to the best of our knowledge, no *completely automatic* BTA for logic programming exists, since all such analyses rely on extra information supplied by the user.

Recently, Mercury was introduced as a logic programming language, specifically tuned towards the creation of large-scale applications [14]. To that extent, the language was provided with a type, mode and determinism system. In this work, we show how precisely this extra information can be fruitfully used in the construction of a completely automatic BTA for Mercury, providing the necessary means for off-line specialisation, suited for industrial application and hence fitting Mercury's design philosophy.

2 Binding-time Analysis

Typical input to a specialiser is a program P and a (partial) description of P's input. Traditionally, this information is described by specifying for each input argument of P whether it is *static* – that is, fully known at specialisation time, or *dynamic* – (possibly) fully unknown at specialisation-time. The primary task of BTA is to obtain such information for *every* program variable [7]. In the context of specialisation of logic programs, classifying program variables as either static or dynamic is in general too coarse-grained (e.g. [3]). We start by describing how Mercury's type system allows us to represent more detailed binding times, enabling the analysis to deal with variables that are *partially static* at specialisation time.

2.1 Describing Binding Times

Mercury is a statically typed language of which we consider a first-order subset. Let $\mathcal{T}_V$, $\mathcal{T}_C$, $\mathcal{T}_F$, Var denote the sets of respectively type variables, type constructors, type functors and program variables for a given program P. A type in Mercury ($\mathcal{M}T$) is either a type variable or a type constructor applied to a number of types:

$$\mathcal{M}T ::= V \in \mathcal{T}_V \mid \gamma(t_1, \ldots, t_n) \text{ with } \gamma/n \in \mathcal{T}_C \text{ and } t_i \in \mathcal{M}T, \forall i \in \{1 \ldots n\}$$

A type constructor $\gamma/n \in \mathcal{T}_C$ is defined by a type definition ($\mathcal{T}Def$), with the constraint that all type variables occurring in the right hand side also occur in the left hand side:

$$\mathcal{T}Def ::= \gamma(V_1,\ldots,V_n) \longrightarrow c_1(t_{1,1},\ldots,t_{1,m_1})\ ;\ldots;\ c_l(t_{l,1},\ldots,t_{l,m_l}).$$
$$\text{where } \gamma/n \in \mathcal{T}_C, c_i \in \mathcal{T}_F, V_i \in \mathcal{T}_V \text{ and } t_{i,j} \in \mathcal{MT}$$

A *type substitution* is defined as a mapping $TSubst : \mathcal{T}_V \mapsto \mathcal{MT}$. By definition, any $t \in \mathcal{MT}$ corresponds with a type tree in the usual way: a type variable $V \in \mathcal{T}_V$ is represented by a single *type node* labelled V, whereas a type $t = \gamma(V_1,\ldots,V_n)\theta$ ($\theta \in TSubst$) is represented by a type node labelled t, which has a *constructor* node labelled c_i as child for each constructor application $c_i(t_{i,1},\ldots,t_{i,n_i})$ in γ's definition. This constructor node in turn has n_i children: the type trees for $t_{i,1}\theta,\ldots,t_{i,n_i}\theta$. Conceptually, a recursive type corresponds to an infinite type tree. However, any type can be represented by a finite *type graph* by imposing an equivalence relation on the type nodes of its type tree: two type nodes on a branch from the root of a type tree are equivalent if they are labelled with the same type. For such a type graph t, $\mathcal{OR}(t)$ represents the (finite) set of type nodes of t.

Example 2.1 The type `list(T)`$\in \mathcal{MT}$ is defined through a type definition `list(T) ---> [] ; [T|list(T)]`, where $T \in \mathcal{T}_V$ and $[]/0$ and $./2 \in \mathcal{T}_F$. See Figure 1 for its type graph.

Given a type graph for $t = \gamma(V_1,\ldots,V_n)\theta$ ($\theta \in TSubst$), $t^{[c_i/k]}$ selects the k-th child of the constructor node labelled c_i among the children of t in the graph. $[c_i/k]$ is called a *selector*, with $[]$ being the empty one: $t^{[]} = t$. In Example 2.1, $list(T)^{[./1]}$ equals T and $list(T)^{[./2]}$ equals $list(T)$ (see Figure 1). Now, we are able to define the notion of an *instantiatedness tree*.

Definition 2.2
An *instantiatedness tree* is a function $\tau : \mathcal{OR}(t) \mapsto \{free, bound\}$ for a given $t \in \mathcal{MT}$ such that $\tau(\eta) = free$ implies that $\tau(\eta') = free$ for all $\eta' \in \mathcal{OR}(t)$ descending from η. The set of all instantiatedness trees over a given type t is denoted by $\mathcal{IT}(t)$; the set of all instantiatedness trees (regardless their type) is denoted by $\mathcal{IT}$.

Any $\tau \in \mathcal{IT}(t)$ represents a set of partial values of type t in which the position corresponding to $\eta \in \mathcal{OR}(t)$ is bound to a functor in case $\tau(\eta) = bound$, and a free variable in case $\tau(\eta) = free$. Figure 1 shows two instantiatedness trees for the type `list(int)`: τ_{il1} denoting a *free* (unbound) value, and τ_{il2}, denoting a list-skeleton: the list constructor is bound, but the elements are free. The domain of instantiatedness trees is partially ordered by the *covers* relation:

Definition 2.3 For any types t_1, t_2 such that $t_1 = t_2\theta$ or $t_2 = t_1\theta$, let $\tau_1 \in \mathcal{IT}(t_1), \tau_2 \in \mathcal{IT}(t_2)$. τ_1 *covers* τ_2, $\tau_1 \succeq \tau_2$ iff $\forall \eta \in \mathcal{OR}(t_1) \cap \mathcal{OR}(t_2), \tau_2(\eta) =$

free implies $\tau_1(\eta) = free$. That is, if $\overline{\tau}_1$, $\overline{\tau}_2$ denote τ_1, respectively τ_2 restricted to $\mathcal{OR}(t_1) \cap \mathcal{OR}(t_2)$, than $\overline{\tau}_1$ can be obtained from $\overline{\tau}_2$ by changing some nodes from *bound* to *free*.

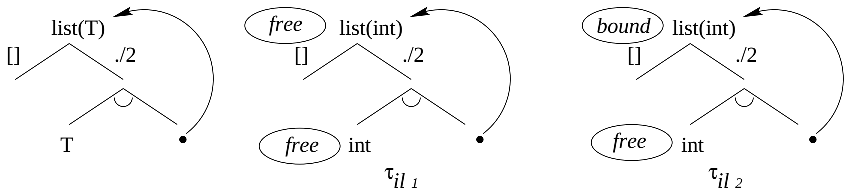

Figure 1: Type graph for `list(T)` and instantiatedness trees for `list(int)`, where $\tau_{il1} \succeq \tau_{il2}$

$\top_t$ (all nodes *free*) and $\bot_t$ (all nodes *bound*) denote the top, respectively bottom elements of $(\mathcal{IT}(t), \succeq)$. In case the type is obvious, we also simply write $\top$ and $\bot$. For $\tau \in \mathcal{IT}(t)$, $\tau^{[c_i/k]}$ denotes an instantiatedness tree $\in \mathcal{IT}(t^{[c_i/k]})$: the subtree of τ consisting of the nodes of $t^{[c_i/k]}$.

Instantiatedness trees are traditionally used in Mercury to define a predicate *mode*: with each argument, say X of type t, a function: $\mathcal{IT}(t) \mapsto \mathcal{IT}(t)$ is associated defining the change in X's instantiatedness state between the predicate's entry and exit points.

In this paper, we adopt the notion of an instantiatedness tree to denote a variable's state of instantiation *during specialisation*. To that extent, we define the notion of a *binding-time environment*:

Definition 2.4 A *binding-time environment* is a mapping $Var \mapsto \mathcal{IT}$. The set of all binding-time environments is denoted by $BTEnv$.

A binding-time environment is thus a set of pairs, associating an instantiatedness tree to a program variable. We use such a set to represent, at a given program point, the instantiatedness state of each of these variables during specialisation.

Example 2.5 Let τ_{il1} and τ_{il2} denote the instantiatedness trees depicted in Figure 1. $\pi = \{(X, \tau_{il1}), (Y, \tau_{il2})\}$ is a binding-time environment stating that the variable X is free during specialisation, whereas the variable Y is bound to a list skeleton.

To ease notation, we allow an element (X, τ) of $\pi \in BTEnv$ to be partially defined, that is, the set $\{(X^{f,1}, \tau_1), \ldots, (X^{f,n}, \tau_n)\}$ defines a single pair (X, τ), where τ is obtained from $\bot_t$ by replacing $\bot_t^{[f/i]}$ by τ_i for $\tau \in \mathcal{IT}(t)$ for all $1 \leq i \leq n$.

2.2 Binding-time Analysis for Mercury

2.2.1 Mercury

A well-typed, well-moded Mercury program [14] consists of a number of procedure definitions. In Mercury terminology, a procedure is a predicate definition with precisely one mode. We consider procedures in which the goals are reordered such that they are mode-correct when evaluated in left-to-right order. Such a procedure can be described by a single clause in *superhomogeneous form* [14]: the arguments in the head of the clause and in predicate calls in the body are replaced by distinct variables, explicit unifications are generated for these variables in the body, and complex unifications are broken down into several simpler ones. Let $Proc$ denote the set of procedures in a given program P.

Definition 2.6 (Superhomogeneous form)

$$\begin{array}{ll} PCall ::= & p(Y_1, \ldots, Y_n) \\ Unif ::= & X \Rightarrow f(Y_1, \ldots, Y_n) \text{ (deconstruction)} \\ & \mid X \Leftarrow f(Y_1 \ldots, Y_n) \text{ (construction)} \mid X == Y \text{ (test)} \\ & \mid X := Y \text{ (assignment)} \\ Atom ::= & PCall \mid Unif \\ Goal ::= & Atom \mid G, A \mid G_1; G_2 \mid if\ G\ then\ A\ else\ A' \mid not\ G \end{array}$$

where $G, G_1, G_2 \in Goal$, $A, A' \in Atom$,[1] $p \in Proc$, $X, Y, Y_1, \ldots, Y_n \in Var$.

Due to the mode correctness, there are a number of additional constraints: the arguments of a predicate call are classified as either input (ground at the time of the call), or output (free at the time of the call, ground at exit). Both variables in a test are of atomic type and input to the test. For the deconstruction defined above, X is input, and all Y_i output, whereas in the construction it is the other way round. To the assignment, X is output and Y input.

A program P consists of N procedure definitions, where each procedure is defined as $p_k(F_1, \ldots, F_{n_k})$:-$G_k$ with $G_k \in Goal$ and $F_1, \ldots, F_{n_k} \in Var$. For each such procedure definition, $\mathcal{F}^k_{in}$ and $\mathcal{F}^k_{out}$ denotes the set of input, respectively output arguments of p_k.

Example 2.7 Consider the definition of $append(list(T) :: in, list(T) :: in, list(T) :: out)$ where for each argument position $t :: in$ or $t :: out$ denotes that the corresponding argument is of type t and is an input, respectively output argument.

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

[1] The goals (G, A) and $if\ G\ then\ A\ else\ A'$ (Definition 2.6) are syntactically more restricted than in Mercury (due to the presence of the atoms). The form of Definition 2.6 can, however, be syntactically derived from the more general one, and enables a more elegant exposition of the analysis.

2.2.2 Basic Binding-Time Analysis

The task of binding-time analysis is, given an instantiatedness state of a main procedure's input arguments, to compute the instantiatedness state of each program variable in a specific atom. Therefore, we need to analyse each procedure's body goal with respect to the binding times for its input arguments. Our analysis is *polyvariant*, that is, different analyses of the same body goal are performed with respect to a different binding-time environment for the procedure's input arguments. Before presenting the actual analysis functions, we define the necessary environments.

Definition 2.8 A *procedure environment* is a mapping $BTEnv \mapsto BTEnv$. The set of all such procedure environments is denoted by $ProcEnv$.

A procedure environment Ψ_k maps, for a procedure p_k, a binding-time environment π_{call} – representing an instantiatedness state for each input argument – to another binding-time environment π_{exit} – representing the corresponding instantiatedness states for the output arguments: $\Psi_k(\pi_{call}) = \pi_{exit}$. In case $\Psi_k(\pi)$ is undefined for $\pi \in BTEnv$, $\Psi_k(\pi) = \{(F, \bot) \mid F \in \mathcal{F}^k_{out}\}$. Reconsider Example 2.7 and τ_{il2} as defined in Figure 1; The procedure environment $\{(\{(X, \tau_{il2}), (Y, \tau_{il2})\}, \{(Z, \tau_{il2})\})\}$ denotes that when *append* is called with both input arguments instantiated to a list skeleton, its output argument will be bound to a list skeleton.

During BTA, we will construct such a procedure environment for every procedure, and collect these in a program environment:

Definition 2.9 A *program environment*, the domain of which is denoted by $ProgEnv$, is an N-tuple of $ProcEnv$, with N the number of procedures in a program P. If $\Psi \in ProgEnv$, Ψ_k denotes the k-th element of Ψ and $\Psi[\Psi_k/\phi]$ denotes Ψ in which Ψ_k is replaced by ϕ: $< \Psi_1, \ldots, \Psi_{k-1}, \phi, \Psi_{k+1}, \ldots, \Psi_N >$ for $\phi \in ProcEnv$.

Starting from the covers relation, we define the following *greater than* relations on the newly introduced environments: $\forall \pi, \pi' \in BTEnv : \pi' \sqsupseteq \pi \Leftrightarrow \forall (X, \tau) \in \pi, (X, \tau') \in \pi'$ and $\tau' \succeq \tau$. For all $\phi, \phi' \in ProcEnv$: $\phi' \sqsupseteq \phi \Leftrightarrow \forall (\pi_i, \pi_o) \in \phi, (\pi_i, \pi'_o) \in \phi'$ and $\pi'_o \sqsupseteq \pi_o$ and $\forall < \Psi_1, \ldots, \Psi_N >$, $< \Psi'_1, \ldots, \Psi'_N > \in ProgEnv$: $< \Psi'_1, \ldots, \Psi'_N > \sqsupseteq < \Psi_1, \ldots, \Psi_N > \Leftrightarrow \forall i \in \{1 \ldots N\} : \Psi'_i \sqsupseteq \Psi_i$. An essential operator is the *least upperbound* of two mappings: When a partial order exists on $\mathcal{R}$, we define for all mappings $f, g : \mathcal{D} \mapsto \mathcal{R}$, $f \sqcup g = \{(d, r_1 \sqcup r_2) \mid (d, r_1) \in f \text{ and } (d, r_2) \in g\} \cup \{(d, r) \mid \text{ either } (d, r) \in f \text{ and } \nexists (d, r') \in g \text{ or } (d, r) \in g \text{ and } \nexists (d, r') \in f\}$, where $\sqcup$ denotes the least upperbound on $\mathcal{R}$. For any set S, $\{e :: S_s\}$ denotes the deconstruction of S such that $S = \{e\} \cup S_s$ and $e \notin S_s$.

We will now introduce the analysis step by step, starting from a first function α which analyses a single atom A, taking the current program environment as input together with a binding-time environment (consisting of binding-times for the variables that are input to A). α updates this binding-time environment with the binding-times for A's output arguments, and possibly adds a new call pattern to the program environment (in case A is a procedure call).

Definition 2.10 $\alpha : \mathit{Atom} \times \mathit{BTEnv} \times \mathit{ProgEnv} \mapsto \mathit{BTEnv} \times \mathit{ProgEnv}$ is defined as

$$
\begin{array}{ll}
\alpha(X \Rightarrow f(Y_1,\ldots,Y_n),\pi,\Psi) & = (\pi \sqcup \{(Y_1,\tau_X^{[f/1]}),\ldots,(Y_n,\tau_X^{[f/n]})\},\Psi) \\
 & \text{ where } (X,\tau_X) \in \pi \\
\alpha(X == Y,\pi,\Psi) & = (\pi,\Psi) \\
\alpha(X := Y,\pi,\Psi) & = (\pi \sqcup \{(X,\tau_Y)\},\Psi) \text{ where } (Y,\tau_Y) \in \pi \\
\alpha(X \Leftarrow f(Y_1,\ldots,Y_n)\pi,\Psi) & = (\pi \sqcup \{(X^{f,1},\tau_{Y_1}),\ldots,(X^{f,n},\tau_{Y_n})\},\Psi) \\
 & \text{ where } \forall i : (Y_i,\tau_{Y_i}) \in \pi \\
\alpha(p_k(X_1,\ldots,X_n),\pi,\Psi) & = (\pi \sqcup \pi_{out},\Psi') \text{ where} \\
 & \pi_{out} = \{(X_i,\tau_{F_i}) \mid F_i \in \mathcal{F}^k_{out}, (F_i,\tau_{F_i}) \in \Psi_k(\pi_{in})\} \\
 & \pi_{in} = \{(F_i,\tau_{X_i}) \mid F_i \in \mathcal{F}^k_{in} \text{ and } (X_i,\tau_{X_i}) \in \pi\} \\
 & \Psi' = \Psi[\Psi_k/\Psi_k \sqcup \{(\pi_{in},\{(F_i,\perp) \mid F_i \in \mathcal{F}^k_{out}\}\}]
\end{array}
$$

Handling unification is straightforward: the binding-time environment π is updated with binding times for the unification's output argument(s), derived from the input binding-times contained in π. Since no new calls are encountered, the program environment Ψ remains unchanged. When analysing a procedure call to p_k, the binding times of p_k's output arguments, given the binding times of its input arguments, are found in Ψ_k, the procedure environment for p_k. The new program environment Ψ' is the old one updated with the procedure environment $(\pi_{in},\{(F_i,\perp) \mid F_i \in \mathcal{F}^k_{out}\})$ of the new call to p_k. The use of $\sqcup$ in the update ensures that if p_k was already analysed with call pattern π_{in}, the resulting exit pattern (already registered in Ψ_k), is preserved and used in the construction of the binding times for the call's actual output arguments. If it was not, the call's output arguments are initialised to $\perp$. The definitions of π_{in} and π_{out} provide the mapping between the formal procedure arguments $(\mathcal{F}^k_{in} \cup \mathcal{F}^k_{out})$ and the actual arguments of the call.

2.2.3 Incorporating Specialisation Control

The binding times produced by BTA reflect a variable's state of instantiatedness during specialisation. Therefore, the bindings for the output variables of an atom A, as produced by $\alpha(A,\pi,\Psi)$, are valid only when A is actually reduced during specialisation and values for A's output variables are computed. When A is residualised, on the other hand, the atom is not evaluated, and no output at all is computed during specialisation. In the latter case, A's output variables – denoted by $\mathit{out}(A)$ – need to be bound to $\top$ during BTA, reflecting their instantiatedness state (free) during specialisation.

To that extent, we introduce a control aspect in the BTA: Our analysis models a rather conservative specialisation control strategy, in the sense that during specialisation, no atoms are reduced that are under dynamic control.

An atom is under dynamic control, when it is not certain to be evaluated at run-time, since it depends on the success or failure of another goal, of which the result is unknown at specialisation-time.

This control strategy is reflected in BTA: The binding times for A's output arguments are made $\top$ (regardless of the result of α) in case A is under dynamic control of a goal G, and reducing G results in residual code of which success or failure at run-time is unknown at specialisation-time.

To model the reducibility of a goal, we introduce the domain $(\mathcal{R}ed, >) = \{\mathbf{RD}, \mathbf{NF}, \mathbf{RS}\}$ on which the ordering $\mathbf{RS} > \mathbf{NF} > \mathbf{RD}$ is imposed. To any goal $G \in Goal$, we associate a value in $\mathcal{R}ed$, denoted by $\mathcal{A}(G, \pi, \Psi)$ specifying the effect of specialising G w.r.t. a binding-time environment π and program environment Ψ: $\mathcal{A}(G, \pi, \Psi) = \mathbf{RD}$ means that G is completely reduced to *true* or *fail* at specialisation-time. $\mathcal{A}(G, \pi, \Psi) = \mathbf{RS}$ denotes that specialising the goal results in some residualised code, and $\mathcal{A}(G, \pi, \Psi) = \mathbf{NF}$ denotes a specialisation result that is either *true* or some residual code, but which is certain not to fail when evaluated at run-time.

Given such a function $\mathcal{A}$, we can define an analysis function β which analyses a goal given a binding-time environment (consisting of binding times for the variables of the goal) and a program environment representing the BTA so far. The result is an updated binding-time environment and an updated program environment.

Definition 2.11 $\beta : Goal \times BTEnv \times ProgEnv \mapsto BTEnv \times ProgEnv$

$$\beta(A, \pi, \Psi) = \alpha(A, \pi, \Psi)$$

$$\beta(not\ G, \pi, \Psi) = \beta(G, \pi, \Psi)$$

$$\beta((G, A), \pi, \Psi) = \begin{cases} (\pi^A, \Psi^A) \text{ if } \mathcal{A}(G, \pi, \Psi) \in \{\mathbf{RD}, \mathbf{NF}\}, \text{ or} \\ (\pi^G \cup \{(V, \top) \mid V \in out(A)\}, \Psi^A) \text{ otherwise} \end{cases}$$
$$\text{where } (\pi^G, \Psi^G) = \beta(G, \pi, \Psi) \text{ and } (\pi^A, \Psi^A) = \alpha(A, \pi^G, \Psi^G)$$

$$\beta((G_1; G_2), \pi, \Psi) = (\pi^{G_1} \sqcup \pi^{G_2}, \Psi^{G_2}) \text{ where } (\pi^{G_1}, \Psi^{G_1}) = \beta(G_1, \pi, \Psi)$$
$$\text{and } (\pi^{G_2}, \Psi^{G_2}) = \beta(G_2, \pi, \Psi^{G_1})$$

$$\beta(if\ G\ then\ A_1\ else\ A_2, \pi, \Psi) = \begin{cases} (\pi^{A_1}, \Psi^{A_1}) \text{ if } \mathcal{A}(G, \pi, \Psi) = \mathbf{NF}, \text{ or} \\ (\pi^{A_1} \sqcup \pi^{A_2}, \Psi^{A_2}) \text{ if } \mathcal{A}(G, \pi, \Psi) = \mathbf{RD}, \text{ or} \\ (\pi^G \sqcup \{(V, \top) \mid V \in out(A_1) \cup out(A_2)\}, \Psi^{A_2}) \end{cases}$$
$$\text{where } (\pi^G, \Psi^G) = \beta(G, \pi, \Psi),\ (\pi^{A_1}, \Psi^{A_1}) = \alpha(A_1, \pi^G, \Psi^G)$$
$$\text{and } (\pi^{A_2}, \Psi^{A_2}) = \alpha(A_2, \pi, \Psi^{A_1})$$

β evaluates a non atomic goal by evaluating its subgoals and combining the resulting binding-time environments, taking the described control strategy into account: When reducing a subgoal G may result in residual code which can possibly fail at run-time, the output variables of an atom A in a conjunction (G, A) are bound to $\top$, since A is under dynamic control of G. Note that code is needed to execute the residual call to A, hence $\alpha(A, \pi^G, \Psi^G)$ is to be computed, and Ψ^A is always the new program environment. The same holds for the atoms A_1 and A_2 in an $if - then - else$ if G reduces to code at specialisation-time. If, on the other hand, G reduces to *true* or to code which is certain not to fail at run-time, only A_1's output binding times are taken into account. If G reduces to $true$ or $fail$ (implying that the if will be reduced at specialisation-time), the output binding times of A_1 and A_2 are combined using $\sqcup$. The result of analysing $not(G)$ is simply the result of analysing G, since in Mercury, a goal $not(G)$ may not bind variables occurring outside G.

Example 2.12 Let G denote the body of the *append* procedure in Example 2.7 and consider the instantiatedness tree τ_{il2} of Figure 1. Let $\pi = \{(X, \tau_{il2}), (Y, \tau_{il2})\}$ be a binding-time environment and Ψ a program environment (with only one entry for *append*), $\Psi =< \emptyset >$. If $(\pi', \Psi') = \beta(G, \pi, \Psi)$, then

$$\pi' = \{(X, \tau_{il2}), (Y, \tau_{il2}), (E, \tau_{il2}^{[./1]}), (E_s, \tau_{il2}), (R, \perp), (Z, \tau_{il2})\}$$
$$\Psi' =< \{(\{(X, \tau_{il2}), (Y, \tau_{il2})\}, \{(Z, \perp)\})\} >$$

Given a goal G, β returns the binding-time environment as computed at the exit of G. In what follows, we sometimes need the binding-time environments computed by β at the entry and exit points of a specific subgoal G' in G, while computing a binding-time environment for G. Therefore, we introduce a variant of β, β': Given G, $G' \in Goal$ with G' a subgoal of G. Let $\beta'(G, G', \pi, \Psi)$ denote a function: $Goal \times Goal \times BTEnv \times ProgEnv \mapsto BTEnv \times BTEnv$ performing exactly the same analysis as β, but returning the binding time environments as computed by β on entry and exit points of G' during analysis of G. Remains the definition of $\mathcal{A}$:

Definition 2.13 $\mathcal{A} : Goal \times BTEnv \times ProgEnv \mapsto \mathcal{R}ed$

$\mathcal{A}(X \Rightarrow f(\ldots), \pi, \Psi) =$ **RS** if $(X, \top) \in BTEnv$, or **RD** otherwise.

$\mathcal{A}(X == Y, \pi, \Psi) =$ **RS** if $(X, \top) \in \pi$ or $(Y, \top) \in \pi$, or **RD** otherwise.

$\mathcal{A}(X := Y, \pi, \Psi) =$ **NF** if $(Y, \top) \in \pi$, or **RD** otherwise.

$\mathcal{A}(X \Leftarrow f(\ldots), \pi, \Psi) =$ **RD**.

$$\mathcal{A}(if\ G\ then\ A_1\ else\ A_2, \pi, \Psi) = \begin{cases} \mathbf{NF} \sqcup \mathcal{A}(A_1, \pi, \Psi) \text{ if } \mathcal{A}(G, \pi, \Psi) = \mathbf{NF}, \text{ or} \\ \mathcal{A}(G, \pi, \Psi) \sqcup \mathcal{A}(A_1, \pi', \Psi) \sqcup \mathcal{A}(A_2, \pi, \Psi) \text{ where}(\pi', \Psi') = \beta(G, \pi, \Psi) \end{cases}$$

$\mathcal{A}(not\ G, \pi, \Psi) =$ **RD** if $\mathcal{A}(G, \pi, \Psi) \in \{\mathbf{RD}, \mathbf{NF}\}$, or **RS** otherwise.

$\mathcal{A}((G_1\ ;\ G_2), \pi, \Psi) = \mathcal{A}(G_1, \pi, \Psi) \sqcup \mathcal{A}(G_2, \pi, \Psi)$.

$\mathcal{A}((G,\ A), \pi, \Psi) = \mathcal{A}(G, \pi, \Psi) \sqcup \mathcal{A}(A, \pi', \Psi)$ where$(\pi', \Psi') = \beta(G, \pi, \Psi)$.

$\mathcal{A}(p_k(X_1, \ldots, X_n), \pi, \Psi) = \mathcal{A}(G_k, \pi', \Psi) \sqcap det(p_k)$
where $\pi' = \{(F_i, \tau_{X_i}) \mid F_i \in \mathcal{F}_{in}^k \text{ and } (X_i, \tau_{X_i}) \in \pi\}$
and G_k the body of p_k's definition.

Whether or not a unification is reducible (**RD**) or should be residualised (**RS**) depends on the binding times of the input variables. Note that an assignment may be residualised, in which case it is classified as non-failing. The reducibility of a composed goal depends on the reducibility of the different subgoals. The reducibility of a procedure call depends on the reducibility for its body goal. In case the body goal is classified as **RS**, the call may be classified **NF** in case the procedure is declared (or inferred) as a non-failing procedure. The determinism declaration of a procedure is formalised by a function $det : Proc \mapsto \mathcal{R}ed$, where $det(p_k) = \mathbf{NF}$ expresses that no call to p_k will fail, whereas $det(p_k) = \mathbf{RS}$ denotes possible failure.

We now return to the basic operation of our BTA: analysing the body of a procedure, with respect to a set of binding times, namely the binding times of the procedure's input arguments for every call pattern collected during analysis. The result is a binding-time environment for the procedure's output arguments, which is recorded in an updated program environment.

Definition 2.14 For any procedure p_i:-G_i with ϕ a procedure environment for p_i and $\Psi \in ProgEnv$, $\delta(G_i, \phi, \Psi) : Goal \times ProcEnv \times ProgEnv \mapsto ProgEnv$ is defined as:

$$\delta(G_i, \phi, \Psi) = \begin{cases} \Psi & \text{if } \phi = \emptyset \\ \delta(G_i, S_s, \Psi'[\Psi'_i/\Psi'_i \sqcup \{(\pi_{in}, \pi'_{out})\}\}) & \text{if } \phi = \{(\pi_{in}, \pi_{out}) :: S_s\} \\ & \text{and } (\pi'_{out}, \Psi') = \beta(G_i, \pi_{in}, \Psi) \end{cases}$$

To analyse a program, its set of procedure bodies needs to be analysed, for which purpose we define Δ, as an extension of δ defined above.

Definition 2.15 For a program p_1:-$G_1, \ldots, p_N$:-G_N, let $S \subseteq \{G_1, \ldots, G_n\}$ and $\Psi \in ProgEnv$. Then $\Delta : 2^{Goal} \times ProgEnv \mapsto ProgEnv$ is defined as:

$$\Delta(S, \Psi) = \begin{cases} \Psi & \text{if } S = \emptyset \\ \Delta(S_s, \delta(G_i, \Psi_i, \Psi)) & \text{if } S = \{G_i :: S_s\} \end{cases}$$

The result of binding-time analysis for a program p_1:-$G_1, \ldots, p_N$:-G_N is the least solution Ψ to the equation $\Psi = \Delta(\{G_1, \ldots, G_N\}, \Psi)$ (1). Starting from an initial program environment Ψ^0, this least solution can be found by repeatedly computing, for all $i > 0$, $\Psi^i = \Delta(\{G_1, \ldots, G_N\}, \Psi^{i-1})$ (2) until a fixpoint is reached. If we assume that p_1 is the program's main procedure and π^1_{in} represents the initial binding-time environment for this procedure's input arguments, $\Psi^0 =< \{(\pi^1_{in}, \{(F, \perp) \mid F \in \mathcal{F}^1_{out}\}), \emptyset, \ldots, \emptyset >$.

2.3 Congruence

The result of BTA, Ψ, represents a complete data flow analysis in terms of the program variables' binding times. In order to be useful, Ψ must be *congruent*: For a variable X of type t and a selector δ, let X^δ denote the part of X's value corresponding to t^δ. If there is a data flow path in P such that $X^{\delta_X} := Y^{\delta_Y}$ (through matching or parameter binding), this path must be represented in Ψ by binding times for X and Y, say τ_X and τ_Y such that $\tau_X^{\delta_X} \succeq \tau_Y^{\delta_Y}$.

Definition 2.16 Given a program p_1:-G_1,...,p_N:-G_N, a program environment $\Psi \in ProgEnv$ is *congruent* when for each atomic subgoal A of any G_i and $\forall(\pi^i_{in}, \pi^i_{out}) \in \Psi_i$ the following holds: If $(\pi_{call}, \pi_{exit}) = \beta'(G_i, A, \pi^i_{in}, \Psi)$ and

- $A = (Y \Rightarrow f(X_1, \ldots, X_n))$ then $\forall i \in \{1 \ldots n\} : \tau_{X_i} \succeq \tau_Y^{[f/i]}$ where (X_i, τ_{X_i}) and $(Y, \tau_Y) \in \pi_{exit}$.
- $A = (Y \Leftarrow f(X_1, \ldots, X_n))$ then $\forall i \in \{1 \ldots n\} : \tau_Y^{[f/i]} \succeq \tau_{X_i}$ where (X_i, τ_{X_i}) and $(Y, \tau_Y) \in \pi_{exit}$.

- $A = (X := Y)$ then $\tau_X \succeq \tau_Y$ where $(X, \tau_X), (Y, \tau_Y) \in \pi_{exit}$.
- $A = p_k(X_1, \ldots, X_n)$ then $\exists(\pi^k_{in}, \pi^k_{out}) \in \Psi_k$ such that
  - $\forall F_i \in \mathcal{F}^k_{in} : \tau_{F_i} \succeq \tau_{X_i}$ where $(F_i, \tau_{F_i}) \in \pi^k_{in}$ and $(X_i, \tau_{X_i}) \in \pi_{call}$.
  - $\forall F_i \in \mathcal{F}^k_{out} : \tau_{X_i} \succeq \tau_{F_i}$ where $(F_i, \tau_{F_i}) \in \pi^k_{out}$ and $(X_i, \tau_{X_i}) \in \pi_{exit}$.

Given this definition of congruence, we can state the following important result:

Theorem 2.17 For a program p_1:-G_1,...,p_N:-G_N, let p_1 be the main procedure and π^1_{in} an initial binding time environment for p_1's input arguments. If $\Psi^0 =< \{(\pi^1_{in}, \{(F, \bot) \mid F \in \mathcal{F}^1_{out}\}), \emptyset, \ldots, \emptyset >$, the fixpoint Ψ^n, $n \in I\!N$, of $\Psi^i = \Delta(\{G_1, \ldots, G_N\}, \Psi^{i-1}), \forall i > 0$ exists and is the least congruent solution to $\Psi = \Delta(\{G_1, \ldots, G_N\}, \Psi)$.

Proof For any update of $\Psi \in ProgEnv$ into Ψ' in α and δ holds that $\Psi \sqsubseteq \Psi'$. By the way the program environment is threaded through Δ via δ, β and α, $\Psi^i \sqsubseteq \Psi^{i+1}$, $\forall i > 0$ in (2). As such, repeated applications of Δ yield a monotonic sequence of program environments, implying the existence of $\Psi \in ProgEnv$ satisfying (1). This fixpoint is finite ($\exists n \in I\!N : \Psi^n = \Psi$): Since for any finite type tree t, $\mathcal{IT}(t)$ is finite, and hence the number of possible program environments for a program P is finite. By definition, α constructs a congruent binding-time environment, and the use of $\sqcup$ in β to combine binding-time environments preserves congruence. So Ψ is a congruent solution. Moreover, it is the least congruent solution, since all exit patterns recorded in Ψ start from $\bot$ and are changed to a higher value (using $\sqcup$) only when congruence so requires. $\square$

3 From Binding Times to Specialisation

3.1 From Binding Times to Annotations

From the program environment resulting from BTA, instructions whether to reduce or residualise a goal can be derived. A suitable way of representing these instructions, is through an annotated version of the source program. To that purpose, we introduce a two-level syntax, as in e.g. [7]: The syntactic domains $2PCall$, $2Unif$, $2Atom$ and $2Goal$ are defined as in Definition 2.6, except that each construct appears in a *static* as well as *dynamic* variant: For a goal $G \in Goal$, G^s and G^d denote the static, respectively dynamic variant of G in $2Goal$. A static annotation is a directive to the specialiser to reduce this goal, a dynamic one the instruction to residualise it. Reducing a statically annotated atom involves either performing unification or unfolding a procedure call – replacing it by a renamed version of the body goal in which parameters have been substituted. Reduction of a static *if then else* consists of evaluating the test goal, and depending on success or failure of the latter, reducing or residualising either the *then* or *else* atom. Reducing a static *not G* involves reducing G, and then either succeeding or failing, depending on the success or failure of G. Conjunctions and disjunctions are not explicitly annotated, since the specialiser treats them as connectives, and always considers both their subgoals for reduction. The two-level versions of procedures ($Proc$) and programs ($Prog$) are denoted by $2Proc$ and $2Prog$.

Definition 3.1 shows how to derive a version of a program P that is annotated with respect to a congruent program environment Ψ. A rule like $\pi \vdash G \ \triangleright_a \ G'$ should be read as follows: G' is a version of G, annotated w.r.t. Ψ, given de binding-time environment π. Given two such rules, ρ_1 and ρ_2, $\frac{\rho_1}{\rho_2}$ denotes that ρ_2 holds if ρ_1 holds. We also introduce the following notation: Given a binding-time environment π, π_G denotes a binding-time environment π_{in} where $(\pi_{in}, \pi_{out}) = \beta'(G_k, G, \pi, \Psi)$ if G_k is the procedure body of which G is a subgoal.

Definition 3.1

$\triangleright_a \ : Goal \mapsto 2Goal, \ \triangleright_a^p \ : Procedure \mapsto 2Procedure, \ \triangleright_a^P \ : Prog \mapsto 2Prog$

$$\frac{\pi \vdash G_1 \triangleright_a G_1', G_2 \triangleright_a G_2'}{\pi \vdash (G_1 \,;\, G_2) \triangleright_a (G_1' \,;\, G_2')} \qquad \frac{\pi \vdash A \in Unif, \mathcal{A}(A,\pi,\Psi) = \mathbf{RD}}{\pi \vdash A \triangleright_a A^s} \qquad \frac{\pi \vdash A \in Unif, \mathcal{A}(A,\pi,\Psi) \in \{\mathbf{NF},\mathbf{RS}\}}{\pi \vdash A \triangleright_a A^d}$$

$$\frac{\pi \vdash G \triangleright_a G', \pi_A \vdash A \triangleright_a A' \text{ and } \mathcal{A}(G,\pi,\Psi) \in \{\mathbf{RD},\mathbf{NF}\}}{(G,A) \triangleright_a (G',A')} \qquad \frac{\pi \vdash G \triangleright_a G' \text{ and } \mathcal{A}(G,\pi,\Psi) = \mathbf{RS}}{(G,A) \triangleright_a (G',A^d)}$$

$$\frac{\pi \vdash G \triangleright_a G' \text{ and } \mathcal{A}(G,\pi,\Psi) = \mathbf{RD}}{\pi \vdash not\ G \triangleright_a not^s\ G'} \qquad \frac{\pi \vdash G \triangleright_a G' \text{ and } \mathcal{A}(G,\pi,\Psi) = \mathbf{RS}}{\pi \vdash not\ G \triangleright_a not^d\ G'} \qquad \frac{\pi \vdash G \triangleright_a G' \text{ and } \mathcal{A}(G,\pi,\Psi) = \mathbf{NF}}{\pi \vdash not\ G \triangleright_a fail}$$

$$\frac{\pi \vdash G \triangleright_a G' \text{ and } \mathcal{A}(G,\pi,\Psi) = \mathbf{RS}}{\pi \vdash if\ G\ then\ A_1\ else\ A_2 \triangleright_a if^d\ G'\ then\ A_1^d\ else\ A_2^d} \qquad \frac{\pi \vdash G \triangleright_a G', \pi_{A_1} \vdash A_1 \triangleright_a A_1' \text{ and } \mathcal{A}(G,\pi,\Psi) = \mathbf{NF}}{\pi \vdash if\ G\ then\ A_1\ else\ A_2 \triangleright_a (G',A_1')}$$

$$\frac{\pi \vdash G \triangleright_a G', \pi_{A_1} \vdash A_1 \triangleright_a A_1', \pi_{A_2} \vdash A_2 \triangleright_a A_2' \text{ and } \mathcal{A}(G,\pi,\Psi) = \mathbf{RD}}{\pi \vdash if\ G\ then\ A_1\ else\ A_2 \triangleright_a if^s\ G'\ then\ A_1'\ else\ A_2'}$$

$$\frac{\pi \vdash p_k(X_1,\ldots,X_n) \in PCall, \exists F_i \in \mathcal{F}_{out}^k : (X_i,\tau) \in \pi \text{ and } \tau \neq \top}{\pi \vdash p_k(X_1,\ldots,X_n) \triangleright_a p_k^c(X_1,\ldots,X_n)^s \text{ where } c = \{\tau_i \mid (X_i,\tau_i) \in \pi \text{ and } F_i \in \mathcal{F}_{in}^k\}}$$

$$\frac{\pi \vdash p_k(X_1,\ldots,X_n) \in PCall, \forall F_i \in \mathcal{F}_{out}^k : (X_i,\top) \in \pi}{\pi \vdash p_k(X_1,\ldots,X_n) \triangleright_a p_k^c(X_1,\ldots,X_n)^d \text{ where } c = \{\tau_i \mid (X_i,\tau_i) \in \pi \text{ and } F_i \in \mathcal{F}_{in}^k\}}$$

$$\frac{\pi \vdash G_i \triangleright_a G_i'}{\pi \vdash p_i(F_1,\ldots,F_n) :- G_i \triangleright_a^p p_i^c(F_1,\ldots,F_n) :- G_i' \text{ where } c = \{\tau_i \mid (F_i,\tau_i) \in \pi \text{ and } F_i \in \mathcal{F}_{in}^i\}}$$

$$\frac{\begin{array}{c}\forall p_i(F_1, \ldots, F_n) : -G_i \in P \text{ and } \forall(\pi_c, \pi_e) \in \Psi_i : \\ \pi_c \vdash p_i(F_1, \ldots, F_n) : -G_i \ \triangleright_a^p \ p_i^c(F_1, \ldots, F_n) : -G_i' \text{ and } p_i^c(F_1, \ldots, F_n) : -G_i' \in P'\end{array}}{P \triangleright_a^P P'}$$

Given the result of BTA, Ψ, for every procedure $p_i(F_1, \ldots, F_{n_i})$:-$G_i \in P$ and for all $(\pi_{call}, \pi_{exit}) \in \Psi_i$, a procedure $p_i^{call}(F_1, \ldots, F_{n_i})$:-$G_i^a$ is created, where p_i^{call} denotes a renaming of p_i w.r.t. π_{call} and G^a is an annotated version of G_i in which every procedure call is renamed according to its call pattern.

3.2 Safeness and Termination of the Specialiser

The rules of Definition 3.1 define a *well-annotated* program [7], which can be shown to be safe: if an atom A is annotated static, the variables that are input to A will be sufficiently instantiated at specialisation-time (as predicted by BTA) in order for A to be reducible. If a *not* G or $if\, G\, then\, A_1\, else\, A_2$ construct is annotated static, G will reduce completely to true or fail at specialisation-time, and hence the construct itself can be evaluated.

Starting from an annotated program and an initial procedure call, say $p_k(t_1, \ldots, t_n)$ with $t_1, \ldots, t_n$ valid Mercury terms, the program is specialised in the usual way: a new procedure $p_k(t_1, \ldots, t_n) : -G'_k$ is created, where G'_k denotes the result of reducing the annotated body of p_k in an environment where p_k's input arguments are bound to the appropriate terms. When during this reduction, code is residualised, the variables in the residualised code are substituted by their value obtained from reduction (if no value for a variable was obtained, it is left as a variable in the residual code). For every procedure call in G'_k, a new procedure is created in the same way, unless it is a variant of a previously encountered call.

Two aspects play a crucial role in termination of this specialisation process: Firstly, the process of reducing a goal should be finite. The defined BTA ensures finiteness of reduction: procedure calls are only unfolded when they are not under dynamic control. As such, termination of the process is guaranteed unless there is a static loop in the program [7]: a static loop meaning that a procedure call is specialised of which an instance does not terminate when evaluated under Mercury. This kind of termination can be compared to the so-called *local* termination in the context of on-line specialisation (e.g. [10]).

Secondly, one must ensure that there is no infinite number of different residual procedure calls to be specialised. This kind of *global* termination is not guaranteed by BTA alone. Consider the following annotated procedure $p(list(T) :: in, T :: in, list(T) :: in, list(T) :: out)$ that deconstructs a list L, and builds a new list R of the same length, but in which all elements are initialised to the value E. The procedure uses an accumulator.

Example 3.2

$$\begin{aligned} p(L, E, Acc, R)&\text{:-} Acc2 \Leftarrow^s [E|Acc], \\ &\quad if^d\, L \Rightarrow^d [X|Xs]\, then\, p(Xs, E, Acc2, R)^d\, else\, R :=^d Acc2. \end{aligned}$$

This procedure is well-annotated given as binding-times that L is *free* and the other input arguments are completely static. Specialising the call $p(L, a, [], R)$ results in an infinite number of residual calls $p(L, a, [], R)$, $p(L, a, [a], R)$, $p(L, a, [a, a], R)$, ... to be specialised.

Global termination can be achieved by specialising appropriate *generalisations* of residualised calls, mapping several residualised calls to the same (specialised) procedure. Since often, this global termination problem arises due to the concrete input to the specialiser, it seems natural to guarantee global termination on the level of the specialiser, when concrete input is available. A wealth of generalisation techniques do exist (see e.g. [10]).

4 Conclusion and Further Work

Performing BTA by abstract interpretation is well studied in the context of functional (e.g. [7, 6, 2]) and imperative languages (e.g. [1]). In [12] for example, a BTA for polymorphically typed languages is presented that is based on *projections* [9], capable of dealing with *partially static* values.

Initial efforts to produce a BTA for logic programming include [13, 8, 3]. These analyses, however, are not completely automatic or lack a general notion of partially static, and are often unable to handle terms that are less instantiated at specialisation-time than they will be at run-time. Much of the encountered difficulties are due to the lack of a type system or the presence of extra-logical features.

In this work, we present a polymorphic and polyvariant BTA for Mercury that, contrary to e.g. [8, 3], exploits the notion of partially static binding-times. We show that the classic framework of BTA by abstract interpretation is applicable to Mercury, but needs to be extended in order to deal correctly with unification and predicates – instead of functions – which can either fail or succeed with multiple output values. The resulting BTA is completely automatic and does not require any more user input than the type and mode information already required by Mercury (except for the initial binding times of the main predicate's input arguments).

The described BTA was implemented in Mercury, and produces, apart from a set of directions for the specialiser, an annotated program encoded in HTML. The use of different colors makes the binding times and the annotations visible for human inspection, illustrating a clear advantage of BTA for specialisation over on-line techniques lacking such a pre-analysis.

Topics of further research include extending the analysis to handle more involved mode declarations and higher-order features of Mercury. A less conservative control strategy – propagating binding-time information over dynamic control structures – may provide better specialisation opportunities at the cost of a specialiser that will have to perform more checks at specialisation-time (becoming more on-line) if it is to guarantee termination. When no strategy at all is incorporated, the resulting analysis is instantiatedness analysis, denoting for every variable its instantiatedness at run-time, given the instantiatedness of the main predicate's input arguments.

Specialisation, combined with other optimisation techniques like structure filtering [4], can change the determinism of a predicate and possibly requires the derivation of new type definitions. Exactly when and how to do this, is also a topic for further research.

Acknowledgments

The authors thank Bern Martens for fruitful discussions, and anonymous referees for valuable comments which helped to improve the paper. Wim Vanhoof is supported by a grant of the IWT, Belgium. Maurice Bruynooghe is supported by "FWO Vlaanderen". This research was partially supported by the ESPRIT ARGo and GOA LP+ projects.

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] J. Gallagher and M. Bruynooghe. Some low-level source transformations for logic programs. In M. Bruynooghe, editor, *Proceedings Meta'90*, pages 229–244, Leuven, April 1990.

[5] J. Gallagher and M. Bruynooghe. The derivation of an algorithm for program specialisation. *New Generation Computing*, 9(3&4):305–333, 1991.

[6] F. Henglein and C. Mossin. Polymorphic binding-time analysis. *Lecture Notes in Computer Science*, 788, 1994.

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

[8] 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.

[9] J. Launchbury. Dependent sums express separation of binding times. In K. Davis and J. Hughes, editors, *Functional Programming*, pages 238 – 253. Springer-Verlag, 1989.

[10] M. Leuschel, B. Martens, and D. De Schreye. Controlling generalisation and polyvariance in partial deduction of normal logic programs. *ACM Transactions on Programming Languages and Systems*, 20(1), 1998.

[11] B. Martens and D. De Schreye. Automatic finite unfolding using well-founded measures. *Journal of Logic Programming*, 28(2):89–146, 1996.

[12] T Mogensen. Binding Time Analysis for Polymorphically Typed Higher Order Languages. In J Diaz and F Orejas, editors, *TAPSOFT'89, Barcelona, Spain*, volume 352 of *LNCS*, pages 298–312. Springer-Verlag, March 1989.

[13] 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.

[14] 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.