Binding-Time Analysis by Constraint Solving
A modular and higher-order approach for Mercury

Wim Vanhoof*

Department of Computer Science, K.U.Leuven, Belgium
e-mail: wimvh@cs.kuleuven.ac.be

Abstract. In this paper we present a binding-time analysis for the logic programming language Mercury. Binding-time analysis is a key analysis needed to perform off-line program specialisation. Our analysis deals with the higher-order aspects of Mercury, and is formulated by means of constraint normalisation. This allows (at least part of) the analysis to be performed on a modular basis.

1 Introduction

Mercury is a recently introduced logic programming language, comprising many 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 several optimizations on the original source code. Although most of these optimizations are implemented as different processes during compilation, some of them are instances of the more general framework of partial evaluation. Examples are inlining of procedure bodies, higher-order call specialisation, specialisation of type-info's [4]. 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 dataflow analysis, and propagate information through the program about what variables are definitely bound to a value at compile-time, independent of the program's runtime input. Such information can be used by a so-called off-line program specialiser to partially evaluate certain parts of the code w.r.t. some given input. An advantage of the more general approach is that the results of BTA can be shared by more than one optimization. As such, different optimizations do not need to redo the analysis, additional optimizations can more

* Supported by the IWT, Belgium.

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.

In a logic programming setting, work on partial evaluation has mainly concentrated on on-line specialisation (where the specialisation process is controlled by the concrete input rather than by a previous analysis [6, 11]). Consequently, little attention has been paid to off-line specialisation and BTA [9, 12, 2]. In previous work [16] we have defined a completely automatic BTA for a subset of the logic programming language Mercury. The current work reformulates and extends our previous work extensively: in contrast with [16], our analysis now deals with the higher-order aspects of Mercury, and its formulation by constraint solving allows it to be performed (at least partially) on the same modular basis as compilation.

Binding-time analysis has been studied thoroughly before in the context of functional languages (e.g. [1, 7, 5]). In this context, the actual binding-time analysis is usually preceded by a flow (or closure) analysis to determine the higher-order control flow in a program (e.g. [8, 13]). In [1], such a flow analysis is combined with a monovariant binding-time analysis by constraint solving. The work in [7, 5] describes a polyvariant BTA by deriving binding-time descriptions that are polymorphic w.r.t. the binding-times of a function's arguments. The described analysis deals with a monomorphic lambda-calculus like language. Techniques exist [10] to deal with more involved type systems and partially static data structures.

This work adapts and generalises ideas from BTA of functional programs in several ways. Basing our domain of binding-times on the type system of Mercury enables us to handle type polymorphism, represent partially static data structures and propagate closure information as part of the binding-time information. Hence our analysis does not require a separate closure analysis, but computes this information using a single set of constraints. Our analysis is polyvariant: it computes different binding-times for the variables in a predicate depending on the binding-times (including the closure information) of the predicate's input arguments. In this work, we develop the basic machinery for such a binding-time analysis for the logic programming language Mercury. Since Mercury is a language specifically tuned towards use in large scale applications, programs written in Mercury usually consists of a number of modules. Hence, performing (at least a large part of) the analysis one module at a time is crucial for such applications. We are closely collaborating with the Mercury developers in Melbourne to implement a BTA based on the presented material in a version of the Mercury compiler which should enable to perform some large scale experiments.

The remainder of the paper is organised as follows: In Sect. 2 we describe the technique of BTA by constraint solving for a first order subset of Mercury, emphasising on the modular approach. In Sect. 3, we alter the technique to deal with the higher-order aspects of the Mercury language.

2 BTA by Constraint Solving: A First-order Setting

Binding-time analysis is about knowing *what* runtime values will be known already at compile-time, without being interested in the values themselves. In order to be useful when complex data structures are involved, the analysis must deal with values that are *partially* known at compile-time. First, we describe a suitable domain for representing such knowledge that was originally introduced in [16]. The analysis itself is presented afterwards.

2.1 A Precise Domain of Binding-Times

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. 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. A possible type graph for this type is denoted in Fig. 1.

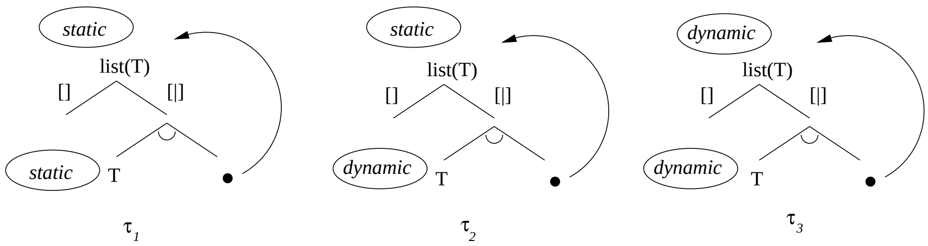

Fig. 1. Labelled type graphs for `list(T)`.

A suitable domain of binding-times is obtained by associating the label *static* or *dynamic* to every or-node in the type graph, the idea being that if a node is labelled *dynamic* respectively *static*, the corresponding subterm(s) in the value described by the binding-time are free, respectively bound to a functor. Also in Fig. 1, τ_1, τ_2 and τ_3 represent such associations denoting, 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.

We first introduce the necessary concepts and notation. The basic Mercury entity our analysis deals with is a compilation unit; that is a Mercury module M together with the definitions from the interfaces of the modules that are imported by M. Throughout the paper, we often simply refer to such a unit as a "module", though. Let $\mathcal{T}_V$, $\mathcal{T}_C$ and $\mathcal{T}_F$ denote the sets of respectively type variables, type constructors and type functors for a given module.

In this section, we consider only first-order types: a type variable or a type constructor applied to a number of types

Definition 1. *Let $\mathcal{MT}$ denote the set of all possible types for a program P:*

$$\begin{array}{l} \mathcal{MT} ::= V \in \mathcal{T}_V \mid \\ \qquad \gamma(t_1, \ldots, t_n) \textit{ with } \gamma/n \in \mathcal{T}_C \textit{ and } t_i \in \mathcal{MT}, \forall i \in \{1 \ldots n\} \end{array}$$

Each type constructor $\gamma/n \in \mathcal{T}_C$ is defined by a type definition in P. The set of all these definitions is denoted by $\mathcal{T}Def$.

Definition 2. *A type constructor is defined as follows, with the constraint that all type variables occurring in the right hand side also occur in the left hand side.*

$$\begin{array}{c} \mathcal{T}Def ::= \gamma(V_1, \ldots, V_n) \longrightarrow c_1(t_{1_1}, \ldots, t_{1_{m_1}})\,; \ldots\,;\, c_l(t_{l_1}, \ldots, t_{l_{m_l}}). \\ \textit{where } \gamma/n \in \mathcal{T}_C, c_i \in \mathcal{T}_F, V_i \in \mathcal{T}_V \textit{ and } t_{i_j} \in \mathcal{MT} \end{array}$$

A *type substitution* is defined as a mapping $TSubst : \mathcal{T}_V \mapsto \mathcal{MT}$. To ease notation, we define the following shorthand notation to denote the set of constructors of a type and a specific subtype of a type: For a type $t \in \mathcal{MT}$, define $C_t = \{c_1/m_1, \ldots, c_l/m_l\} \subseteq \mathcal{T}_F$ and $ST_t^{c_i,k} = t_{i,k}\theta$ if $t = \gamma(V_1, \ldots, V_n)\theta$ with $\theta \in TSubst$ and $\gamma(V_1, \ldots, V_n) \longrightarrow c_1(t_{1,1}, \ldots, t_{1,m_1})\,; \ldots\,;\, c_l(t_{l,1}, \ldots, t_{l,m_l}) \in \mathcal{T}Def$.

Example 1. The type `list(T)` introduced above is formally defined by the following definition:

```
:- type list(T) ---> [] ; [T | list(T)].
```

Given the type $list(T)$, $C_{list(T)} = \{[]/0, [|]/2\}$ and $ST_{list(T)}^{[|],1} = T$ and $ST_{list(T)}^{[|],2} = list(T)$.

In what follows, we represent a *sequence* over a set S by $\langle S \rangle$; $\langle\rangle$ denotes the empty sequence, and $\langle d_1, \ldots, d_n\rangle \bullet \langle e_1, \ldots, e_m\rangle$ denotes the concatenation of the two sequences, resulting in the new sequence $\langle d_1, \ldots, d_n, e_1, \ldots, e_m\rangle$. For any sequence S, S_i denotes the i'th element of S, if it exists.

We will often need to refer to subvalues of values, according to a type tree. Since a node in a type tree is uniquely defined by a path from the root towards it, we define the notion of a *type path* as a sequence over $\mathcal{T}_F \times \mathbb{N}$. The set of all type paths is denoted by *TPath*. Type paths are used to select (type) nodes in a type graph: For a type t, $t^{\langle\rangle} = t$ and $t^{\langle(c,k)\rangle \bullet \delta}$ denotes $t_{c,k}^{\delta}$ if $c \in C_t$ and $t_{c,k} = ST_t^{c,k}$.

Given the notion of type paths, we can define a (possibly infinite) type tree in a straightforward way:

Definition 3. *For a type $t \in \mathcal{MT}$, the* first order type tree *of t, $\mathcal{L}_t \in 2^{TPath}$ is recursively defined as:*

- $\langle\rangle \in \mathcal{L}_t$
- *if $t = \gamma(t_1, \ldots, t_n)$ with $\gamma \in \mathcal{T}_C$ then $\forall c/n \in C_t, k \in \{1, \ldots, n\}$, $\langle(c,k)\rangle \bullet \delta \in \mathcal{L}_t$ where $\delta \in \mathcal{L}_{t_{c,k}}$ and $t_{c,k} = ST_t^{c,k}$.*

Recursive type definitions correspond to infinite type trees. However, to consider finite type *graphs*, it is sufficient to impose an equivalence relation on the set of type paths for a given type, turning it into a finite set. We consider two type paths δ, δ' to be *equivalent* w.r.t a type t if and only if $\delta = \delta' \bullet \gamma$ (for some $\gamma \in TPath$) and $t^{\delta} = t^{\delta'}$. Let $\equiv$ denote the transitive closure of the *equivalent* relation and $\mathcal{L}_t^{\equiv}$ denotes a type tree modulo $\equiv$.

Example 2. For $list(T)$ as defined in Example 1, $\mathcal{L}_{list(T)}^{\equiv} = \{\langle\rangle, \langle([\,|\,], 1)\rangle\}$, since the path referring to the recursive occurrence of $list(T)$, $\langle([\,|\,], 2\rangle$ is equivalent with $\langle\rangle$.

If we introduce the domain $\mathcal{B} = \{static, dynamic\}$, we can define a binding-time for a type t by a labelling of $\mathcal{L}_t^{\equiv}$'s nodes as follows:

Definition 4. *A* binding-time *for a type $t \in \mathcal{MT}$ is a function $\tau : \mathcal{L}_t^{\equiv} \mapsto \mathcal{B}$ such that $\forall \delta \in dom(\tau) : \tau(\delta) =$* dynamic *implies that $\tau(\delta') =$* dynamic *for all $\delta' \in dom(\tau), \delta' = \delta \bullet \epsilon$. The set of all binding-times (regardless their type) is denoted by $\mathcal{BT}$.*

We impose the ordering *dynamic* $>$ *static* on $\mathcal{B}$ which induces an ordering on $\mathcal{BT}$: $\tau_1 \geq \tau_2$ if and only if $\tau_1(\delta) \geq \tau_2(\delta)$ for all $\delta \in TPath$. In Fig. 1, for example, $\tau_3 > \tau_2 > \tau_1$.

2.2 Generating a Constraint System

The task of BTA can be described as follows: given binding-times for a predicate's input arguments, compute binding-times for the predicate's remaining variables. In [16], 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 and often leads to duplication of analysis efforts. In this work, we reformulate BTA in a constraint-solving setting, enabling an efficient modular approach: First, the data flow inside a predicate is examined resulting in a number of constraints on the variables' binding-times. These constraints are created without taking the call pattern of interest into account. Next, the least solution to this constraint system, combined with the call pattern of interest, provides a binding-time for each variable that is both correct and as much static as possible.

First, we show how to translate a module to a constraint system. Such a module can be considered as a set of *procedures*, which are obtained by translating the original predicates and functions to superhomogeneous form [15]. The definition of a procedure is converted to 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 is either an atom or a number of goals connected by *conjunction*, *disjunction*, *if then else* or *not*. An atom is either a unification or a procedure call.

Definition 5.

$$\begin{array}{rl} Proc ::= & p(H_1,\ldots,H_n) : -G \ \mathit{with}\ p \in \mathcal{T}_P, G \in Goal, H_1,\ldots,H_n \in \mathcal{V}ar \\ Goal ::= & Atom \mid not\ G \mid (G_1,G_2) \mid (G_1;G_2) \mid if\ G_1\ then\ G_2\ else\ G_3 \\ & \mathit{with}\ G, G_1, G_2, G_3 \in Goal \\ Atom ::= & X = f(Y^1,\ldots,Y^n) \mid X = Y \mid p(Y^1,\ldots,Y^n) \\ & \mathit{with}\ X, Y, Y^1,\ldots,Y^n \in \mathcal{V}ar, f \in \mathcal{T}_F\ \mathit{and}\ p \in \mathcal{T}_P \end{array}$$

An essential part of compiling a Mercury module is *mode analysis* [15]. During this analysis, unifications are split into four different types, according their data flow: *test* denoted by $X == Y$ (where X and Y are both input and of atomic type - that is having a type tree consisting of only the root node), *assignment* denoted by $X := Y$ (where Y is input, X is output), *construction* denoted by $X \Leftarrow f(Y^1,\ldots,Y^n)$ (where X is output, $Y^1,\ldots,Y^n$ input and $f \in \mathcal{T}_F$) and *deconstruction* denoted by $Y \Rightarrow f(X^1,\ldots,X^n)$ (where Y is input, $X^1,\ldots,X^n$ output and $f \in \mathcal{T}_F$). For a procedure p, $\mathcal{A}rg(p)$ denotes the sequence $\langle H_1,\ldots,H_n\rangle$ of p's formal arguments. Each atom in a procedure's body is uniquely identified by a *program point* (PP).

Example 3. Consider the predicate $append(list(T) :: in, list(T) :: in, list(T) :: out)$ 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. The predicate is defined as follows, where each atom is subscribed with a natural number, representing its program point.

$$\begin{array}{ll} \{append(X,Y,Z)\}_0\text{:-} & \{X \Rightarrow []\}_1, \{Z := Y\}_2; \\ & \{X \Rightarrow [E \mid Es]\}_3, \{append(Es,Y,R)\}_4, \{Z \Leftarrow [E|R]\}_5. \end{array}$$

Variables can be initialised in different branches of a disjunction (or if-then-else). To improve precision, we associate different binding-times to such variables, one for each occurrence in such a branch. Notationally we distinguish between such occurrences by subscribing a variable with the program point (represented by a natural number) where it got initialised. In the *append* example, Z_2 denotes the occurrence of Z in the first branch of the disjunction, whereas Z_5 denotes the occurrence of Z in the second branch. We identify the head of a procedure with program point "0" and use this program point to denote input arguments and (final occurrences) of output arguments. The set of all occurrences of variables is denoted by $\mathcal{V}ar_{pp}$.

When computing *the* binding-time of a variable, it is mandatory to take the right (sub)set of its occurrences into account. We therefore introduce the following notions: An *execution path* in a predicate p is a sequence of program points in p denoting a possibly nonfailing derivation of the atoms associated to the program points. Two program points p_1 and p_2 *share an execution path* in p if there exists an execution path S in p such that both p_1 and p_2 are elements of S. We define the function $init : \mathcal{V}ar \times PP \mapsto 2^{\mathcal{V}ar_{pp}}$ as $init(X, pp) = \{X_{pp'} \mid X$ got initialised at pp' and pp and pp' share an execution path$\}$.

Example 4. In the *append* example, the only two execution paths are $\langle 0,1,2\rangle$ and $\langle 0,3,4,5\rangle$. We have, for example, that $init(Y,2)=\{Y_0\}$, $init(X,1)=\{X_0\}$, $init(E,5)=\{E_3\}$ and $init(R,5)=\{R_4\}$.

The binding-times that can be associated to (an occurrence of) a variable are constrained by the binding-times of those variables it is related to through a data flow relation. To express data flow relations in terms of binding-times, we introduce the notion of a *binding-time constraint.*

Definition 6. *A* binding-time constraint *is denoted as* $X_{pp}^{\delta} \geq S$, *where* $X_{pp} \in \mathcal{V}ar_{pp}$, $\delta \in TPath$ *and* $S = \text{static} \mid \text{dynamic} \mid Y_{pp'}^{\gamma}$ *(with* $Y_{pp'} \in \mathcal{V}ar_{pp}$ *and* $\gamma \in TPath$*). The set of all binding-time constraints is denoted by* $\mathcal{B}_{\mathcal{C}}$.

Due to the well-modedness of Mercury procedures, it is possible to trace the dataflow back to the procedure's input arguments, and consequently to express the binding-time constraints on a variable in function of the binding-times of the procedure's (input) arguments. Binding-time constraints in this format are said to be in *normal form.*

Definition 7. *A binding-time constraint* $X_{pp}^{\delta} \geq S$ *is in* normal form *if its right-hand side* S *is either* static, dynamic *or* Y_0^{γ} *with* Y *being an input argument.*

In particular, we are often interested in the normalised binding-time constraints on a procedure's output arguments. For a procedure p, we denote with p's *normal form* a set of binding-time constraints, where the left-hand side is an (output) argument of p. We start by defining how to translate a procedure into a set of binding-time constraints. A unification can straightforwardly be translated to a set of binding-time constraints on the involved variables, whereas translating a call to a set of binding-time constraints requires the called predicate's normal form. If we denote with μ a function $Proc \mapsto 2^{\mathcal{B}_{\mathcal{C}}}$ such that for a procedure q, $\mu(q)$ denotes q's normal form, we can define for any atom A its associated set of binding-time constraints as $\mathcal{A}_{\mu}(A)$, where $\mathcal{A}$ is defined as follows:

Definition 8. *Consider an atom at program point pp.* $\mathcal{A}_{\mu}(A)$ *is as follows, depending on the type of atom:*

$$
\begin{array}{ll}
\mathcal{A}_{\mu}(X == Y) & = \{\} \\
\mathcal{A}_{\mu}(X := Y) & = \{X_{pp} \geq Y_{pp_Y} \mid Y_{pp_Y} \in init(Y,pp)\} \\
\mathcal{A}_{\mu}(X \Leftarrow f(Y^1,\ldots,Y^n)) = & \left\{ \begin{array}{l} \{X_{pp} \geq \text{static}\} \cup \\ \{X_{pp}^{\langle (f,1)\rangle} \geq Y_{pp_{Y^1}}^1 \mid Y_{pp_{Y^1}}^1 \in init(Y^1,pp)\} \\ \cup \ldots \cup \\ \{X_{pp}^{\langle (f,n)\rangle} \geq Y_{pp_{Y^n}}^n \mid Y_{pp_{Y^n}}^n \in init(Y^n,pp)\} \end{array} \right. \\
\mathcal{A}_{\mu}(Y \Rightarrow f(X^1,\ldots,X^n)) = & \left\{ \begin{array}{l} \{X_{pp}^1 \geq Y_{pp_Y}^{\langle (f,1)\rangle} \mid Y_{pp_Y} \in init(Y,pp)\} \\ \cup \ldots \cup \\ \{X_{pp}^n \geq Y_{pp_Y}^{\langle (f,n)\rangle} \mid Y_{pp_Y} \in init(Y,pp)\} \end{array} \right. \\
\mathcal{A}_{\mu}(p(X^1,\ldots,X^n)) & = \mathcal{C}_{\mu(p)}(p(X^1,\ldots,X^n))
\end{array}
$$

where $\mathcal{C}_N(p(X^1,\ldots,X^n))$ *is defined as*

$$\begin{array}{l} \{X^{i^\delta}_{pp} \geq X^{j^\gamma}_{pp_j} \mid X^j_{pp_j} \in \mathit{init}(X^j, pp) \text{ and } F^{i^\delta}_0 \geq F^{j^\gamma}_0 \in N\} \\ \cup \{X^{i^\delta}_{pp} \geq c \mid F^{i^\delta}_0 \geq c \in N \text{ with } c = \text{static or dynamic}\} \\ \text{where } \langle F^1, \ldots F^n\rangle = \mathcal{A}rg(p) \end{array}$$

Due to the well-modedness of procedures, the set $\mathit{init}(Y, pp)$ in Definition 8 contains program points that precede the unification in the procedure's body. Naturally, an atom constrains only the binding-time(s) of its output variable(s) in order to comply with the *congruence* [8] condition: if a (subvalue of) a variable is constructed using (a subvalue of) another variable, then the corresponding subvalue in the former's binding-time should be at least as dynamic as the corresponding part in the latter's binding-time. Note that a construction with a constant, $X \Leftarrow c$, leads to a (superfluous) constraint $X \geq \mathit{static}$ whereas a deconstruction with a constant $X \Rightarrow c$ leads to no constraints at all. The function $\mathcal{C}_{\mu(p)}$ maps the normal form of p's binding-time constraints – expressed in terms of the formal arguments F^i_0 – to the actual arguments of the call to p: F^i_0 in a left-hand side occurrence is replaced by X^i_{pp}; F^i_0 in a right-hand side occurrence is replaced by $X^i_{pp_i}$ with pp_i the program point(s) where X^i gets initialised.

Now, we are in a position to define the set of constraints associated to a procedure:

Definition 9. *Given* $\mu : Proc \mapsto 2^{\mathcal{B}c}$ *such that* $\mu(q)$ *denotes* q*'s normal form, we define for a procedure* $p(F^1,\ldots,F^n) : -G$

$$BTC_p(\mu) = \mathcal{F} \cup \bigcup_{\text{atom } A\in G} \mathcal{A}_\mu(A)$$

where $\mathcal{F} = \{F^i_0 \geq F^i_{pp} \mid F^i_{pp} \in \mathit{init}(F^i, 0) \text{ and } F^i \text{ an output argument}\}$

Note that in the above definition, $\mathcal{F}$ links the final binding-time of p's output arguments to each occurrence of that argument in p's body where it is initialised.

Example 5. If we consider $\mu(\mathit{append}) = \{\}$ for the *append* procedure from Example 3, $BTC_{app}(\mu)$ is defined as:

$$\begin{array}{cccc} Z_0 \geq Z_2 & Z_2 \geq Y_0 & E_3 \geq X_0^{\langle([|],1)\rangle} & Z_5^{\langle([|],1)\rangle} \geq E_3 \\ Z_0 \geq Z_5 & & Es_3 \geq X_0 & Z_5 \geq R_4 \end{array}$$

Bringing a set of binding-time constraints to normal form can be achieved by repeatedly unfolding the variable in the right-hand side, replacing it with the right-hand sides of the constraints that exist on that value. Unfolding is complicated by the use of subvalues, since a variable may need to be unfolded w.r.t. constraints that define a subvalue as well as a supervalue of the former. If we consider two subvalues of a variable, say X^δ and X^γ, we know that one of them is a subvalue of the other if either δ is an extension of γ or vice versa.

Definition 10. *We define* $ext : TPath \times TPath \mapsto TPath \times TPath$ *as follows:*

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

and undefined otherwise.

Note that if $ext(\gamma,\delta) = (\alpha,\beta)$ then $\gamma \bullet \alpha = \delta \bullet \beta$. Unfolding a constraint $X^{\gamma}_{pp_X} \geq Y^{\delta}_{pp_Y}$ w.r.t. a set of constraints results in a set of new constraints on (possible subvalues of) $X^{\gamma}_{pp_X}$, with as right hand sides the appropriate subvalues of the right hand sides of the constraints that were used for unfolding. To denote a subvalue of a constraint's right-hand side S, we use the notation $S^{\bullet\eta}$. If S denotes the variable X^{γ}_{pp}, then $S^{\bullet\eta}$ equals $X^{\gamma\bullet\eta}_{pp}$. Otherwise, if S denotes the constant *static* or *dynamic*, $S^{\bullet\eta}$ simply equals S.

$$project(X^{\gamma}_{pp_X} \geq Y^{\delta}_{pp_Y}, \mathcal{D}) = \{\, X^{\gamma\bullet\eta_1}_{pp_X} \geq S'^{\bullet\eta_2} \mid Y^{\delta'}_{pp_Y} \geq S' \in \mathcal{D} \text{ and } ext(\delta,\delta') = (\eta_1,\eta_2)\}$$

Note that unfolding results in an empty set when the right-hand side that is unfolded is a variable on which no constraints are defined. Normalising a constraint set then consists of repeatedly selecting a constraint that is not yet in normal form and replacing it with the constraints resulting from unfolding, until a fixed point is reached.

Definition 11. *We define* $compr_p : 2^{\mathcal{B}c} \mapsto 2^{\mathcal{B}c}$ *as follows:*

$$\begin{array}{rl} compr_p(\mathcal{D}) = & \mathcal{D} \setminus \{X^{\gamma}_{pp_X} \geq Y^{\delta}_{pp_Y} \mid X^{\gamma}_{pp_X} \geq Y^{\delta}_{pp_Y} \in \mathcal{D} \ \textit{and}\ pp_Y \neq 0\} \\ & \cup\, project(X^{\gamma}_{pp_X} \geq Y^{\delta}_{pp_Y}, \mathcal{D}) \end{array}$$

and use the classical notation to apply $compr_p$ *a number of times starting with a set* $\mathcal{D}$*:* $compact_p(\mathcal{D}) = compr_p \uparrow \omega$ *where*

$$\begin{array}{l} compr_p \uparrow 0 = \mathcal{D} \\ compr_p \uparrow i = compr_p(compr_p \uparrow (i-1))\ \forall i > 0 \end{array}$$

Computing the normal form of the predicates defined in a module M is done through a fixed point computation. Note that we compute, for each procedure p, a set of binding-time constraints that is larger than p's normal form, as we are interested in the normalised binding-time constraints on all local variables of p. The result of the process for a module M is denoted by a function $\mu_M : Proc \mapsto 2^{\mathcal{B}c}$, mapping a procedure to a set of normalised binding-time constraints. For such a function μ_M, we denote with $\mu_{M_{|o}}$ the restricted mapping, where for each p, $\mu_{M_{|o}}(p) \subseteq \mu_M(p)$ and $\mu_{M_{|o}}(p)$ denotes p's normal form. When the analysis for the module M starts, we assume that all procedures defined in other modules have been analysed before, the result recorded in a function μ_I. If they are not, μ_I is initialised with *dynamic* for all the output arguments.[1] Construction of μ_M

[1] See [3] for a discussion of how a inter module analysis can cope with cyclic dependencies between modules.

is then defined as a fixed point iteration over an operator $T_\mu : (Proc \mapsto 2^{\mathcal{B}c}) \mapsto (Proc \mapsto 2^{\mathcal{B}c})$ defined as follows:

$$\begin{array}{l} T_\mu(S) = S \setminus \{(p,N) \mid (p,N) \in S \text{ and } p \text{ defined in } M\} \\ \qquad\quad \cup \{(p, compact_p(BTC_p(S_{|o})))\} \end{array}$$

The analysis starts from an initial mapping where each predicate of M is mapped onto an empty set of constraints. During each iteration round, the constraints from a procedure's body are generated and normalised. This process is repeated until a fixed point is reached.

Definition 12. *For a module M, the result of constraint generation is the mapping $\mu_M = T_\mu \uparrow \omega$, where $T_\mu \uparrow \omega$ is the fixed point of*

$$\begin{array}{l} T_\mu \uparrow 0 = \mu_I \;\cup\; \{(p, \{\}) \mid p \in Proc \textit{ defined in } M\} \\ T_\mu \uparrow k = T_\mu(T_\mu \uparrow (k-1)) \; \forall k \geq 1 \end{array}$$

Example 6. Let M_{app} denote a module consisting only of the definition of *append*. The result of constraint generation, for the *append* procedure, is the following set of binding-time constraints:

$$\begin{array}{lllll} Z_0 \geq Y_0 & Z_2 \geq Y_0 & E_3 \geq X_0^{\langle([\,|\,],1)\rangle} & R_4 \geq Y_0 & Z_5^{\langle([\,|\,],1)\rangle} \geq X_0^{\langle([\,|\,],1)\rangle} \\ Z_0^{\langle([\,|\,],1)\rangle} \geq X_0^{\langle([\,|\,],1)\rangle} & & Es_3 \geq X_0 & R_4 \geq X_0^{\langle([\,|\,],1)\rangle} & Z_5 \geq Y_0 \end{array}$$

It can be proven that the fixed point $\mu_M = T_\mu \uparrow \omega$ is reached after a finite number of steps. Informally, since binding-time constraints reflect the data flow relations in a *well-moded* procedure, no circular dependencies exist in the set and hence unfolding is finite and results in a finite set of binding-time constraints in normal form. New constraints are incorporated in each iteration round, but due to monotonicity of *compact* and the fact that the number of possible binding-time constraints is finite for a procedure, the iteration process results in a fixed point.

2.3 Solving the Cconstraint System

Given a set of binding-time constraints in normal form for a procedure and a call pattern of interest for that procedure, it is straightforward to compute the binding-time of an occurrence of a variable in that procedure. Let us formally define a call pattern (*Callp*) for a procedure p/n as a sequence over $\mathcal{BT}$ of length n. To compute binding-times, we define the function $\beta : Proc \times Callp \times 2^{\mathcal{B}c} \times \mathcal{V}ar_{pp} \mapsto \mathcal{BT}$ as follows:

Definition 13. *We define $\beta : Proc \times Callp \times 2^{\mathcal{B}c} \times \mathcal{B} \mapsto \mathcal{BT}$ as follows: For a program variable X of type t_X that is initialised at program point pp,*

$$\begin{array}{rll} \beta(p,\pi,\mathcal{D},X_{pp}) = & \textit{if} & X = \mathcal{A}rg(p)_i \textit{ and input then } \pi_i \\ & \textit{else} & \{(\delta, \text{static}) \mid \delta \textit{ a TPath} \in t_X\} \\ & & \sqcup\{(\gamma \bullet \eta, c) \mid \; X^\gamma_{pp} \geq c \in \mathcal{D}, c \in \{\text{static}, \text{dynamic}\} \textit{ and} \\ & & \qquad\qquad \eta \textit{ a TPath} \in t^\gamma_X\} \\ & & \sqcup\{(\gamma \bullet \eta, \tau(\eta)) \mid \; X^\gamma_{pp} \geq Y_0^{i^\delta} \in \mathcal{D}, \eta \textit{ a TPath} \in t^\gamma_X \textit{ and} \\ & & \qquad\qquad \tau = \pi^\delta_i\} \end{array}$$

Note that computing binding-times using β is a cheap process, as it merely consists of looking up some binding-times from the call pattern and combining (subvalues of) these using the least upperbound operator on $\mathcal{BT}$. The resulting binding-time is the *least dynamic* one satisfying the constraints from $\mathcal{D}$.

Example 7. Reconsider $\mu_{M_{app}}$ from Example 6 and the binding-times as depicted in Fig. 1. Given the call pattern $\pi = \langle \tau_2, \tau_1, \bot \rangle$,

$$\begin{aligned}
\beta(app, \pi, \mu_{M_{app}}, Z_0) = \; & \{(\langle\rangle, static), (\langle([|], 1)\rangle, static)\} \\
& \sqcup \{(\langle\rangle, static), (\langle([|], 1)\rangle, static)\} \\
& \;\; \text{from } Z_0 \geq Y_0 \\
& \;\; \text{and } \beta(p, \pi, \mathcal{D}, Y_0) = \{(\langle\rangle, static), (\langle([|], 1)\rangle, static)\} \\
& \sqcup \{(\langle([|], 1)\rangle, dynamic)\} \\
& \;\; \text{from } Z_0^{\langle([|],1)\rangle} \geq X_0^{\langle([|],1)\rangle} \\
& \;\; \text{and } \beta(p, \pi, \mathcal{D}, X_0) = \{(\langle\rangle, static), (\langle([|], 1)\rangle, dynamic)\} \\
= \; & \{(\langle\rangle, static), (\langle([|], 1)\rangle, dynamic)\} = \tau_2
\end{aligned}$$

In this section, we have described BTA for a first-order subset of Mercury as a 2-phase process. The first, and computationally most involved phase – the computation of a set of normalised binding-time constraints – is performed *independent* of any call pattern of interest. Hence, this phase needs to be run only once for each module; its results can be recorded and used when analysing other modules that import this one. It is only the second phase – computing binding-times w.r.t. a call pattern – that needs to be repeated for every call and call pattern of interest.

Example 8. Consider a module M_{rev}, importing the module M_{app} and consisting of the definition of reverse: $rev(list(T) :: in, list(T) :: out)$.[2]

$$\begin{aligned}
rev(X, R) :- \; & X \Rightarrow [\,], R \Leftarrow [\,]; \\
& X \Rightarrow [E|Es], rev(Es, Rs), X' \Leftarrow [E], append(Rs, X', R).
\end{aligned}$$

When construction $\mu_{M_{rev}}$, the constraints for the call to *append* are renamings of $\mu_{M_{app}}(append)$ (See Example 6) w.r.t. the mapping $\{X \mapsto Rs, Y \mapsto X', Z \mapsto R\}$:

$$R^{\langle([|],1)\rangle} \geq Rs^{\langle([|],1)\rangle} \qquad\qquad R \geq X'$$

resulting (without re-analysing *append* itself) in the following constraints for *rev*:

$$\begin{array}{lll}
E \geq X^{\langle([|],1)\rangle} & X'^{\langle([|],1)\rangle} \geq X^{\langle([|],1)\rangle} & R \geq static \\
Es \geq X & X' \geq static & R^{\langle([|],1)\rangle} \geq X^{\langle([|],1)\rangle}
\end{array}$$

[2] In the remaining examples, we associate only one occurrence with each variable and leave out the program point subscription in order to ease notation.

3 BTA in a Higher-order Setting

Mercury is a higher-order language in which *closures* can be created, passed as arguments of predicate calls, and in turn be called themselves. In this section, we reconsider the defined BTA in such a higher-order context.

To deal with the higher-order issues in Mercury, it suffices to extend the definition of superhomogeneous form (see Definition 5) with two new kinds of atoms: A *higher-order unification*, $X \Leftarrow p(V^1, \ldots, V^k)$ with $p \in \mathcal{T}_P$ and $V^1, \ldots, V^k \in \mathcal{V}ar$, constructs a closure from a predicate p where $V^1, \ldots, V^k$ are the curried arguments. Closures are called using a *higher-order call* $X(V^{k+1}, \ldots, V^n)$ where $V^{k+1}, \ldots, V^n \in \mathcal{V}ar$ are the closure's arguments. To express higher-order binding-times, we also consider a *higher-order type* to be included in the set of types ($\mathcal{MT}$). A higher-order type is a type definition of a predicate like $p(t_1, \ldots, t_n)$ with $t_i \in \mathcal{MT}$. When constructing a type tree, higher-order types are considered leaf nodes (the argument types are thus not taken into account in the type tree).

3.1 Closure Information

The basic problem when analysing a procedure involving higher-order calls, is that the control flow in the procedure is determined by the value of the higher-order variables. Consequently, without knowing (an approximation of) these values, it is impossible to compute meaningful data dependencies between the procedure's variables. Consider the following example:

Example 9. The *map* predicate converts the first list of type T to a new list of type T using the predicate provided in the second argument.

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

In this example, we know from the mode declaration that E_1 is input and E_2 is output from the call to P. To compute a meaningful binding-time for E_2 (and all variables depending on E_2), we need to consider the dependencies that most likely exist between E_2 and E_1. Obviously, this requires information on the possible closures P can be bound to. Without this information, we can only approximate E_2's binding-time by *dynamic*, resulting in the following constraints for *map*:

$$
\begin{array}{lll}
L_0 \geq static & L_0 \geq Es_2 & L_0^{\langle([\,],1)\rangle} \geq Es_2\\
E_1 \geq L_1^{\langle([\,],1)\rangle} & Es_1 \geq L_1 & E_2 \geq dynamic
\end{array}
$$

However, if there is in the module a call to *map* that binds P for example to the predicate *rev* from above, then we can create a more precise set of constraints for *map w.r.t. the fact that* $P = rev$, and the call $P(E_1, E_2)$ can, during constraint generation, be treated as a first-order call $rev(E_1, E_2)$, resulting in the following extra constraints (which are a renaming from $\mu_{M_{rev}}(rev)$, see Example 8):

$$E_2 \geq static \qquad E_2^{\langle([],1)\rangle} \geq E_1^{\langle([],1)\rangle}$$

Normalising and incorporating the constraints for the call to *map* now results in the following set of constraints:

$$\begin{array}{lll}
E_1 \geq L_1^{\langle([],1)\rangle} & E_2^{\langle([],1)\rangle} \geq L_1^{\langle([],1),([],1)\rangle} & L_0 \geq static \\
Es_1 \geq L_1 & E_2 \geq static & L_0^{\langle([],1)\rangle} \geq static \\
 & & L_0^{\langle([],1),([],1)\rangle} \geq L_1^{\langle([],1),([],1)\rangle}
\end{array}$$

Informally, the constraints on *map*'s output argument L_0 express that in the least solution, the binding-times of the elements of the output lists will be the same as the binding-times of the elements of the input lists.

3.2 Higher-order BTA

The general idea behind performing the analysis in a higher-order setting is to associate different sets of constraints to the same predicate, depending on the closures occurring in its call pattern.

To make this information available during BTA, we extend the notion of a binding-time to include, for higher-order variables, a set of closures. This set approximates the specific closure bound to the variable at runtime. For analysis purposes, we denote such a closure with $p(\tau_1, \ldots, \tau_k)$ where p is a procedure and $\tau_1, \ldots, \tau_k$ are binding-times for the curried arguments. The set of all such closures is denoted by $\mathcal{C}los$. In what follows, we extend the necessary definitions from Sect. 2. First, we extend the domain $\mathcal{B}$ by an explicit bottom element $\perp$ and elements *static*(S) with $S \in 2^{\mathcal{C}los}$. The domain is ordered by the lattice in Fig. 2.

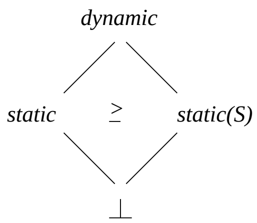

Fig. 2. The ordering on $\mathcal{B}$

The ordering between elements *static*(S_1) and *static*(S_2) in turn is determined by the lattice $\{2^{\mathcal{C}los}, \supseteq\}$, that is: *static*$(S_1) \geq$ *static*(S_2) if and only if $S_1 \supseteq S_2$. Given this notion of $\mathcal{B}$, the definition of a binding-time remains unchanged.

In the higher-order BTA, we associate a set of binding-time constraints to a call/call pattern pair and consequently denote the result of analysis by a function $\mu^c : \mathit{Proc} \times \mathit{Callp} \mapsto 2^{\mathcal{B}_{\mathcal{C}}}$. Like in the first-order case, the entries in μ^c contain the set of normalised binding-time constraints of the involved procedure/call pattern pair and we denote with $\mu^c_{|o}$ the restriction of these sets to the procedure's normal form (under the involved call pattern).

Computing the set of binding-time constraints associated to a procedure p w.r.t. call pattern π requires the normal forms of the procedures called in the body of p. Hence, we denote this set with $BTC_p^{\pi}(\mu^c)$, which is defined like BTC_p in the first-order case, except for the handling of procedure calls.

Like before, handling a first-order call involves renaming the called predicate's normal form, which now depends on that call's call pattern. Let $q(X^1, \ldots, X^n)$ denote a call in procedure p's body at program point pp. The constraints associated with this call are $\mathcal{C}_{\mu^c(q,\langle\tau_1,\ldots,\tau_n\rangle)}(q(X^1, \ldots, X^n))$ where $\mathcal{C}$ is defined as in Definition 8 and $\langle\tau_1, \ldots, \tau_n\rangle$ denotes the call pattern of the call, which is computed using the set of binding-time constraints already associated with p and π: $\forall i : \tau_i = \beta(p, \pi, \mu^c(p, \pi), X^i_{pp_i})$.

Handling a higher-order call $P(X^{k+1}, \ldots, X^n)$ is a bit more complicated. First of all, the binding-time of P is computed in the current environment (that is, the procedure p and call pattern π). If this binding-time turns out to be *dynamic* (indicating that P is not bound to a set of closures at specialisation-time), the binding-times of the call's output arguments are simply approximated by *dynamic*, by introducing the following constraints for the call:

$$\{X^i_{pp} \geq \mathit{dynamic} \mid X^i \text{ is an output argument}\}$$

If, on the other hand, the binding-time of P turns out to be $\mathit{static}(S)$ – S denoting the set of closures P can be bound to, the higher-order call can be treated as a number of first-order calls (one for each closure in S), and its set of associated constraints is the union of the (renamed) normal forms associated to each of the derived first-order calls:

$$\bigcup_{q(\tau_1,\ldots,\tau_k)\in S} \mathcal{C}_{\mu^c_{|o}(q,\langle\tau_1,\ldots,\tau_n\rangle))}(q(X^{k+1}, \ldots, X^n))$$

Note that the involved call pattern $\langle\tau_1, \ldots, \tau_n\rangle$ is a combination of the binding-times from the closure and the binding-times of the remaining arguments (computed as in the first-order case). The constraints associated to $q(\langle\tau_1, \ldots, \tau_n\rangle)$ are renamed with respect to the sequence of variables

$$\langle P^{\langle(pred,1)\rangle}, \ldots, P^{\langle(pred,k)\rangle}, X^{k+1}, \ldots, X^n\rangle$$

where $P^{\langle(pred,j)\rangle}$ denotes a new binding-time variable that refers to the j-th binding-time from the closure associated to P.

Example 10. Consider a higher-order call $P(L_2, L_3)$ and suppose we know that P is bound to a closure $\mathit{append}(\tau_1)$ (the *append* predicate from Example 3 with its first argument curried). This call introduces the following constraints, by appropriate renaming of *append*'s normal form:

$$L_3 \geq L_2 \qquad L_3^{\langle([\,],1)\rangle} \geq P^{\langle(pred,1)\rangle}$$

3.3 Towards Maximum Modularity

Like in the first-order case, when analysing a module M, we start from an initial table μ_I^c, containing the analysis results for the imported modules. We return to the issue of modularity further on. An entry for a call and call pattern (p, π) of interest is added to the table, by computing $BTC_p^\pi(\mu_I^c)$ resulting in a new table in which (empty) entries are added for all call/call pattern pairs for which, during computation of $BTC_p^\pi(\mu_I^c)$, the normal form needed to be retrieved but was not yet available. The set of normalised binding-times is then (re)computed for each entry in the table belonging to a procedure in M, possibly requiring to add new entries to the table. This process is repeated until a fixed point is reached.

It is important to note that, when constructing the table, a new entry for p w.r.t. call pattern π should only be created if there is not yet an entry for (p, π') in the table, where π' is a call pattern of which the *higher-order parts* (i.e. the involved closures) are equal to those of π. For constraint generation, only these higher-order parts are significant, since the set of generated constraints does not depend on the first-order parts of the call pattern.

The analysis of a module starts from $(p_i, \langle dynamic, \ldots, dynamic\rangle)$ for every exported predicate p_i. Using $\langle dynamic, \ldots, dynamic\rangle$ as a call pattern ensures that no (higher-order) information from outside the module is assumed and thus the result of analysis, μ_M^c, can be used when analysing other modules that import M. Higher-order information that is *local* to a module, however, is used if BTA deals with higher-order unification. Therefore, we extend the notion of a binding-time constraint so that its right-hand side now can include $static(p(Y^1, \ldots, Y^k))$, as we associate the constraints

$$\{X \geq static(p(Y^1_{pp_1}, \ldots, Y^k_{pp_k})) \mid Y^1_{pp_1} \in init(Y^1, pp), \ldots, Y^k_{pp_k} \in init(Y^k, pp)\}$$

with such a higher-order unification $X \Leftarrow p(Y^1, \ldots, Y^k)$. The minimal binding-time for X satisfying this constraint says that X is bound to a closure created from the predicate p and binding-times for $Y^1, \ldots, Y^k$.

The right-hand side of such a constraint is "constant", and it can readily be used by β to compute a binding-time for X. All β needs to do is to compute the binding-times $\tau_1, \ldots, \tau_k$ for $Y^1, \ldots, Y^k$ (in the same environment) and guarantee that if the binding-time for X turns out to be $static(S)$ (by evaluating possibly other constraints on X), that the closure $p(\tau_1, \ldots, \tau_k) \in S$. Note however that while such a constraint can be considered to be in normal form (as it can readily be used by β), it can only be considered as part of the normal form of the procedure it is part of, if the arguments of the closure construction are arguments of that procedure. If not, such a constraint can not be directly renamed into a caller's environment.

Consider the following example, defining a predicate $lrev(list(list(T)) :: in, list(list(T)) :: out)$ that reverses each of the lists in its first argument:

$$lrev(L_1, L_2) :- P \Leftarrow rev, map(L_1, P, L_2).$$

When analysing *lrev* w.r.t. the call pattern $\langle dynamic, dynamic \rangle$, the constraint $P \geq static(rev())$ is derived for *lrev* in a first iteration round. In a next round, this constraint is used to derive $\langle dynamic, static(\{rev()\}), dynamic \rangle$ as call pattern for the call to *map*. If *map* and *rev* are defined in the same module as *lrev*, a specific version of the constraints for *map* is created, leading to the following set of normalised constraints for *lrev* (the constraints with only *static* in the right-hand side are removed):

$$\{L_2^{\langle([|],1),([|],1)\rangle} \geq L_1^{\langle([|],1),([|],1)\rangle}\}$$

Informally, this constraint implies that in the least solution the binding-times of the elements of the output lists will be the same as the binding-times of the elements of the input lists. Note that this set of constraints is independent of the call pattern of *lrev*, as this does not contain any higher-order information.

If *map* and/or *rev* are defined in a different module (say M'), then there are several possibilities to follow. A first one is to rename the constraints associated to a version of *map* w.r.t. a call pattern that is more dynamic. Note that this option is always available, as *map* – being an exported predicate of M' – will at least be analysed w.r.t. $\langle dynamic, \ldots, dynamic \rangle$, the result recorded in μ_I^c. While correct, this option will not result in useful constraints for predicates using *map*.

A more interesting option is to make sure that when analysing M, enough information is available such that *map* can be (re)analysed w.r.t. this new call pattern, outside the scope of the earlier analysis of M'. Note that doing so, might require to (re)analyse all procedures involving higher-order arguments imported by *map* and other procedures imported therein. Recent work [3] indicates how such a call dependent analysis can still be performed on a modular basis, reanalysing one module at a time and propagating analysis results – possibly triggering reanalysis of other modules, until ”enough” precision is obtained.

4 Discussion

In this work, we have rephrased a BTA for Mercury [16] using constraint normalisation, an approach that allows – in contrast with our earlier work – to perform (a large part of) the analysis on a modular basis. When no higher-order control flow is involved, constraint normalisation can be performed one module at a time, bottom-up in the module graph. To obtain maximal precision with predicates that do involve higher-order flow, it can be necessary to re-analyse the predicate with respect to specific closure information in the predicate's call pattern. Dealing with modular programs is mandatory to apply program analysis tools on real-world programs. Recently, analysis of modular programs gained attention in a logic programming setting. For example, [14] discusses some – mainly practical – issues in the analysis of modular programs.

The current work formalises the ideas presented in [17] and extends the analysis to deal with the higher-order concepts of Mercury. In our analysis, closures are encapsulated in the notion of a binding-time, and our analysis does not require a separate closure analysis. Closure information from the call pattern is incorporated during constraint generation and normalisation. The analysis is polyvariant in this respect, as different such call patterns result in different constraint sets for the same predicate. Computing concrete binding-times is straightforward given binding-times for a predicate's input arguments, as it merely consists of computing the least solution of the predicate's associated normal form. The analysis is also polyvariant in this respect: different call patterns result in a different least solution of the same constraint set.

We are closely collaborating with the Mercury developers in Melbourne to implement the described BTA in a version of the Mercury compiler. To that extent, the tight interaction between the binding-time analysis and a concrete specialiser (ensuring, for example, that binding-times of a call's output arguments are treated as *dynamic* when the specialiser would not unfold the call) needs to be modeled. This can be achieved by adding constraints to the system that model the conditions under which atoms are evaluated at specialisation-time. Topics of further work are to perform some large scale experiments with the analysis, and couple it with the partial evaluation mechanisms of the compiler.

Acknowledgments

The author likes to thank Maurice Bruynooghe for fruitful discussions on the subject, and anonymous referees for their valuable feedback. Special thanks go to Zoltan Somogyi for providing the opportunity of working together with the Mercury team, and 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 author's visit to Australia an unforgettable experience.

References

1. A. Bondorf and J. Jørgensen. Efficient analyses for realistic off-line partial evaluation. *Journal of Functional Programming*, 3(3):315–346, July 93.
2. 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.
3. F. Bueno, M. de la Banda, M. Hermenegildo, K. Marriott, G. Puebla, and P. Stuckey. A model for inter-module analysis and optimizing compilation. In *Preproceedings of LOPSTR 2000*, 2000.

4. Tyson Dowd, Zoltan Somogyi, Fergus Henderson, Thomas Conway, and David Jeffery. Run time type information in Mercury. In *Proceedings of the International Conference on the Principles and Practice of Declarative Programming*, volume 1702 of *Lecture Notes in Computer Science*, pages 224–243. Springer-Verlag, 1999.
5. D. Dussart, F. Henglein, and C. Mossin. Polymorphic recursion and subtype qualifications: Polymorphic binding-time analysis in polynomial time. In A. Mycroft, editor, *International Static Analysis Symposium, Glasgow, Scotland, September 1995, (Lecture Notes in Computer Science, vol. 983)*, pages 118–136. Berlin: Springer-Verlag, 1995.
6. J. Gallagher. Specialisation of logic programs: A tutorial. In *Proceedings PEPM'93, ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–98, Copenhagen, June 1993. ACM Press.
7. F. Henglein and C. Mossin. Polymorphic binding-time analysis. *Lecture Notes in Computer Science*, 788, 1994.
8. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
9. 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.
10. J. Launchbury. Dependent sums express separation of binding times. In K. Davis and J. Hughes, editors, *Functional Programming*, pages 238 – 253. Springer-Verlag, 1989.
11. 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.
12. 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.
13. Jens Palsberg. Closure analysis in constraint form. *ACM Transactions on Programming Languages and Systems*, 17(1):47–62, January 1995.
14. 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.
15. 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.
16. 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.
17. 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.