Binding-time Analysis for Mercury

Wim Vanhoof[1]⋆, Maurice Bruynooghe[2], and Michael Leuschel[3]

[1] University of Namur
wva@info.fundp.ac.be
[2] Katholieke Universiteit Leuven
Maurice.Bruynooghe@cs.kuleuven.ac.be
[3] University of Southampton
mal@ecs.soton.ac.uk

Abstract. In this work, we develop a binding-time analysis for the logic programming language Mercury. We introduce a precise domain of binding-times, based on the type information available in Mercury programs, that allows the analyser to reason with partially static data structures. The analysis is polyvariant, and deals with the module structure and higher-order capabilities of Mercury programs.

1 Introduction

Program specialisation is a technique that transforms a program into another program, by precomputing some of its operations. Assume we have a program P of which the input can be divided in two parts, say s and d. If one of the input parts, say s, is known at some point in the computation, we can *specialise* P with respect to the available input s. This specialisation process comprises performing those computations of P that depend only on s, and recording their *results* in a new program, together with the *code* for those computations that could not be performed (because they rely on the input part d – unknown at this point in the computation). The result of the specialisation is a new program, P_s that computes, when provided with the remaining input part d, the *same* result as P does when provided with the complete input $s + d$. Comprising a mixture of program evaluation and code generation, the program specialisation process is also often referred to by the names *partial evaluation*, *mixed computation* or *staged computation*.

Staging the computations of a program can be useful (usually in terms of efficiency) when different parts of a program's input become known at different times during the computation. The best benefit can be obtained when a single program must be run a number of times while a part of its input remains constant over the different runs. In this case, the program can first be specialised with respect to the constant part of the input, while afterwards the resulting program can be run a number of times, once for each of the remaining (different) input

⋆ Sponsored by FWO Vlaanderen and Katholieke Universiteit Leuven at the time of this research.

parts. In such a staged approach, the computations that depend only on the constant input part are performed only once – during specialisation. In the non-staged approach, *all* computations – including those depending on the constant part – are performed in every run of the program.

When using program specialisation to stage the computations of a program, the basic problem is deciding what computations can be safely performed during the specialisation process. The driving force behind this decision is twofold. Firstly, the specialisation process itself must terminate; that is, the specialiser must not to get into a loop when evaluating a sequence of computations from the program that is to be specialised. Secondly, the obtained degree of specialisation should be "as good as possible", meaning that a fair amount of computations that *can* be performed during specialisation *are* effectively performed during specialisation.

The key factor determining whether a computation can be performed during specialisation is the fact whether enough input values are available to compute a result. If that is the case, the specialiser can perform the computation; if not, it should generate code to perform this computation at a later stage. Binding-time analysis is a static analysis that, given the program and a description about the available partial input with respect to which the program will be specialised, computes for every statement in the program what input values will be known when that statement is reached during specialisation. In addition, the analysis computes — according to some control strategy — whether or not the statement should be evaluated during specialisation.

Once the program P and its available partial input s has been analysed by binding-time analysis, specialisation of P with respect to s boils down to evaluating those statements in P that are annotated as such by the binding-time analysis. This specialisation technique is called *offline*, the reason being that most of the control decisions about what statements should be evaluated have been taken by the binding-time analysis. This contrasts with the so-called *online* specialisation technique in which the program to be specialised is not analysed by any binding-time analysis, but is directly evaluated with respect to its partial input under the supervision of a control system that decides – for every statement under consideration – on the fly whether or not it can safely be evaluated. Both approaches towards specialisation have their advantages and disadvantages. In this work, we concentrate on *offline* specialisation and construct a binding-time analysis for the logic programming language Mercury.

1.1 Binding-time Analysis and Logic Programming

Using binding-time analysis to control the behaviour of the specialisation has been thoroughly investigated in a number of programming paradigms. Breaking work on offline program specialisation of imperative languages include C-mix by Andersen [1] and more recently Tempo [10, 20] by Consel and his group. In the context of functional language specialisation, most work focusing on binding-time analysis and offline specialisation was originally motivated by the desire to achieve better self-application [13, 24]. Whereas initial analysis dealt with first-order languages [24], more recently developed analyses deal with higher-order aspects [15, 4], polymorphism [32, 19] and partially static data structures [28].

In the field of logic programming, however, only little attention has been paid to offline program specialisation. Known exceptions are LOGIMIX [33] and LOGEN [25] that develop different approaches to offline program specialisation for Prolog. Both cited works, however, lack an automatic binding-time analysis and rely on the user to provide the specialiser with suitable annotations of the program. To the best of our knowledge, the only attempt to construct an automatic binding-time analysis for logic programming is [6] and our own work about which we report in [30]. The approach of [6] is particular, in the sense that it obtains the required annotations not by analysing the subject program directly but rather by analysing the behaviour of an *online* program specialiser on the subject program. Although conceptually interesting, the latter approach is overly conservative and restricts the number of computations that can be performed during specialisation. Indeed, [6] decides whether to unfold a call or not based on the original program, not taking current annotations into account. This means that a call can either be completely unfolded or not at all. The binding-time analysis first described in [50] and employed in [30] is also particular in the sense that it obtains its annotations by repeatedly applying an automatic termination analysis. If the termination analysis identifies a call as possibly non-terminating, that call is marked such that it will not be reduced by the specialiser. Then the termination analysis is rerun to prove termination of the program under the assumption that each call that is marked as non-reducible is not evaluated. The process is repeated until termination of the (annotated) program can be proven.

Both the approach of [6] and [30] have been designed towards dealing with untyped and unmoded logic programming languages. The fact that most logic programming languages are untyped makes it harder to represent the availability of *partial* input in a sufficiently precise way during the analysis. More importantly, the lack of control flow information in the program makes it very difficult to approximate the data flow in a sufficiently precise way and renders the derivation of a binding-time analysis by ”classic” abstract interpretation techniques not straightforward, hence the approaches of [6] and [30]. In this work, we construct a completely automatic binding-time analysis for the recently introduced logic programming language Mercury. Being a strongly typed and moded language, Mercury lifts the obstacles encountered in more traditional logic programming languages and allows to construct a “traditional” binding-time analysis along the lines of [15, 23] based on data flow analysis. However, the more involved data- and control flow features – inherent to a logic programming language – render the derivation of an automatic binding-time analysis a daunting and not straightforward task.

1.2 Mercury

The design of Mercury was started in October 1993 by researchers at the University of Melbourne. While logic programming languages had been around for quite some time, no one seemed to fully realise the theoretical advantages such a language would have over more traditional, imperative languages. These advantages are widely known, and are summarised for example in [42]: a higher level of expressivity (enabling the programmer to concentrate on *what* has to be done rather than on *how* to do it), the availability of a useful formal semantics (required for the – relatively – straightforward design of analysis and transformation tools), a semantics that is independent of any order of evaluation (useful for parallelising the code), and a potential for declarative debugging [31]. While a language like Prolog does offer some of these advantages, others are destroyed by the impure features of the language.

The main objective of the Mercury designers was to create a logic programming language that would be *pure* and useful for the implementation of a large number of *real-world* applications. To achieve this goal, the main design objectives of Mercury can be summarised as follows [42]: *Support for the creation of reliable programs.* This involves a language that allows the compiler to detect some classes of bugs. *Support for programming in teams.* Large software systems are usually build by a number of programmers. The language must provide good support for creating a single application from multiple parts that are build (sometimes in isolation) by different programmers. These two objectives form a major departure from Prolog which, at the time, had basically no support for programming in the large, and which does not allow a lot of type-, mode- and determinism errors to be caught at compile-time. Another important objective was *support for the creation of efficient programs.* The compiler had to produce code whose performance is competitive with that produced by compilers of other languages. To meet these design objectives, Mercury was fitted with a strong system of type-, mode- and determinism declarations. Besides providing the programmer with some valuable documentation, these declarations enable the compiler to check the internal consistency of the program and to spot a substantial number of bugs that would go unnoticed in declaration free code submitted to a Prolog compiler. Also, the availability of declarations allows to adapt the evaluation order of the body atoms in a predicate and provides as such the basis for an efficient execution mechanism of the language [11, 41, 43]. Mercury is equipped with a modern module system that enables to hide some data definitions and to encapsulate both data and code, and provides as such support for programming-in-the-large activities.

1.3 Structure of the paper

The remainder of this paper is organised as follows. In the following section, we introduce a domain of binding-times that is based on the type information available in Mercury programs. Next, in Section 3, we introduce a 2-phase binding-time analysis for a first-order subset of Mercury. The first part of the analysis performs a symbolic data flow analysis that – being call-independent – can be performed for each module in isolation, bottom-up over the module hierarchy. The second phase of the analysis, which computes the actual annotations, is call-dependent by nature and relies on the result of the symbolic analysis for all modules involved. In Section 4, we lift the first-order restriction and enhance the analysis such that it computes and propagates closure information throughout the program that is being analysed. In Section 5, we work through an example and discuss to what extent our method is also applicable to typed Prolog programs. We conclude this paper in Section 6 with a discussion of our binding-time analysis and its relation with existing work in the literature.

2 A Domain of Binding-times

Binding-time analysis can be seen as an application of abstract interpretation over a domain of *binding-times*. A binding-time abstracts a value by specifying at what time during a 2-stage computation[1] the value becomes known. In their most basic form, the binding-time of a value is either *static* or *dynamic*, denoting a value that is known early, during specialisation, or late, during evaluation of the residual program, respectively.

It is recognised [23] that for a logic programming language, approximating values by either *static* or *dynamic* is too coarse grained in general. Indeed, most logic programs use a lot of *structured* data, where data values are represented by structured terms. Consequently, the input to the specialiser usually consists of a partially instantiated term: a term that is less instantiated than it would be at run-time. Approximating a partially instantiated term by *dynamic* usually results in too much information loss, possibly resulting in missed specialisation opportunities. Therefore, we use the structural information from the type system of Mercury to represent more detailed binding-times, capable of distinguishing between the computation stages in which *parts* of a value (according to that value's type) become known.

In what follows, we formally define the notions of type, type definition, type trees and type graphs, which we wil use later on as the basis of our abstract domain. Our formalisation is mainly based on [47, 48] and [46], but similar notions and definitions can be found in related work on program analysis involving types, like e.g. [39, 22, 45, 38, 37]. Mercury's type system is based on a polymorphic many-sorted logic, and corresponds to the Mycroft-O'Keefe type system [34]. Basically, the types are discriminated union types and support parametric polymorphism: a type definition can be parametrised with some type variables, as the following example in Mercury syntax shows.

Example 1. `:- type list(T) ---> [] ; [T | list(T)].`

The above defines a polymorphic type `list(T)`: it defines values of this type to be terms that are either `[]` (the empty list) or of the form `[A|B]` where `A` is a value of type `T` and `B` is a value of type `list(T)`.

Formally, if we denote with $\Sigma_{\mathcal{T}}$ the set of type constructors and with $V_{\mathcal{T}}$ the set of type variables of a language $\mathcal{L}$, the set of *types* associated to $\mathcal{L}$ is

[1] Generalisations exist in which computations are staged over more than 2 stages (see e.g. [14]). In this work, we focus on a traditional 2-stage process, dividing the computations in a program over *specialisation-time* versus *run-time*.

represented by $\mathcal{T}(\Sigma_{\mathcal{T}}, V_{\mathcal{T}})$; that is the set of terms that can be constructed from $\Sigma_{\mathcal{T}}$ and $V_{\mathcal{T}}$. A type containing variables is said to be *polymorphic*, otherwise it is a *monomorphic* type. A *type substitution* is a substitution from type variables to types. The application of a type substitution to a polymorphic type results in a new type, which is an *instance* of the original type.

As usual, the set of program values is denoted by $\mathcal{T}(\mathcal{V}, \Sigma)$; that is the set of terms that can be constructed from a set Σ of function symbols and a set $\mathcal{V}$ of program variables.

The relation between a type and the values (terms) that constitute the type is made explicit by a *type definition* that consists of a number of *type rules*, one for every type constructor. Example 1 shows the type rule associated to the `list/1` type constructor. Formally, a type rule is defined as follows:

Definition 1 (type rule). *The* type rule *associated to a type constructor $h/n \in \Sigma_{\mathcal{T}}$ is a definition of the form*

$$h(\overline{T}) \rightarrow f_1(\overline{\tau}_1) \; ; \; \ldots \; ; \; f_k(\overline{\tau}_k).$$

where $\overline{T}$ is a sequence of n type variables from $V_{\mathcal{T}}$ and for $1 \leq i \leq k$, $f_i/m \in \Sigma$ with $\overline{\tau}_i$ a sequence of m types from $\mathcal{T}(\Sigma_{\mathcal{T}}, V_{\mathcal{T}})$ and all of the type variables occurring in the right hand side occur in the left hand side as well. The function symbols $\{f_1, \ldots, f_k\}$ are said to be associated with the type constructor h. A finite set of type rules is called a type definition.

Given a type substitution, we define the notion of an instance of a type rule in a straightforward way. In theory, every type (constructor) can be defined by a type rule as above. In practice, however, it is useful to have some types builtin in the system. For Mercury, the types `int`, `float`, `char`, `string` are builtin types whose denotation is predefined and is the set of integers, floating point numbers, characters and strings respectively. A type is called *atomic* if it is defined by a set of zero-arity function symbols $\{f_1, \ldots, f_k\}$.

Mercury is a statically typed language, in which the (possibly polymorphic) type of every term occurring in the program text is known at compile-time. In what follows, we use the type definition to construct, for every type occurring in the program, a finite description of the *structure* that values belonging to the denotation of a particular type can take. The relevance of such a description is in the fact that it can be used to abstract the values belonging to the denotation of the type according to their structure. This allows the construction of a precise abstract domain for program analysis, in particular binding-time analysis.

To extract a structural description of a type from a type definition, we introduce the notion of a type-path being a sequence of functor/argument position pairs that is meant to denote a path through the type definition from a type to an occurrence of one of its subtypes. In fact, a type itself can be represented as a (possibly infinite) set of such paths, one for every path from the type that is being defined to some subtype occurring at a particular position within some term belonging to the denotation of that type. More formally, we denote the set of all such sequences over $\Sigma \times \mathbb{N}$ by *TPath*. The empty sequence is denoted by $\langle\rangle$, and given $\delta, \gamma \in \mathit{TPath}$, we denote with $\delta.\gamma$ the sequence obtained by concatenating γ to δ. A *type tree* for a particular type can then be defined as follows:

Definition 2 (type tree). *Given a type $\tau \in \mathcal{T}(\Sigma_{\mathcal{T}}, V_{\mathcal{T}})$, the* type tree *of τ, denoted by $\mathcal{L}_\tau$, is a set of sequences from TPath and is recursively defined as:*

- $\langle\rangle \in \mathcal{L}_\tau$
- *if $\tau = h(\overline{T})\theta$ and $h(\overline{T}) \rightarrow f_1(\overline{\tau}_1); \ldots; f_k(\overline{\tau}_k)$ is a type rule then $\langle(f_i, j)\rangle.\delta \in \mathcal{L}_\tau$ where (i) $i \in \{1 \ldots k\}$, (ii) f_i has arity m in Σ, (iii) $j \in \{1 \ldots m\}$, (iv) τ_{i_j} denotes the j-th type in $\overline{\tau}_i$, and (v) $\delta \in \mathcal{L}_{(\tau_{i_j})\theta}$.*

Note that the type tree of an atomic type is $\{\langle\rangle\}$ as a term belonging to an atomic type does not have any subterms. Likewise, also the type tree of a type variable T is defined as $\mathcal{L}_T = \{\langle\rangle\}$.

Example 2. Reconsider the type *list*(T) from Example 1. As $\mathcal{L}_T = \{\langle\rangle\}$, the type tree of *list*(T) is the infinite set of type paths

$$\mathcal{L}_{\mathit{list}(T)} = \left\{ \begin{array}{l} \langle\rangle \\ \langle([|], 1)\rangle \\ \langle([|], 2)\rangle \\ \langle([|], 2), ([|], 1)\rangle \\ \langle([|], 2), ([|], 2)\rangle \\ \langle([|], 2), ([|], 2), ([|], 1)\rangle \\ \langle([|], 2), ([|], 2), ([|], 2)\rangle \\ \langle([|], 2), ([|], 2), ([|], 2), ([|], 1)\rangle \\ \ldots \end{array} \right\}$$

The general idea now is to define, for any type τ, a finite approximation of $\mathcal{L}_\tau$ that provides a good characterisation of the structure of terms of type τ. First we introduce the following notation that formally defines the (sub)type that is identified by a type-path within another type.

Definition 3 (type selected by type-path). *Let $\tau = h(\overline{T})\theta$ be a type and δ a path in $\mathcal{L}_\tau$. If $\delta = \langle\rangle$ then $\tau^\delta = \tau$. Otherwise, δ has the form $\langle(f, i)\rangle.\gamma$, the type rule for $h(\overline{T})$ has in the right-hand side an alternative of the form $f(\tau_{i_1}, \ldots, \tau_{i_{k_i}})$ and $\tau^\delta = \tau_{j_i}^\gamma$.*

Note that a type path $\delta \in \mathcal{L}_\tau$ can also be used to identify a particular subterm in a term $t : \tau$, if it exists. Indeed, if $\delta \in \mathit{TPath}$ is of the form $\delta = \langle(f, i)\rangle.\gamma$ and $t = f(t_1, \ldots, t_n)$ we define $t^\delta = t_i^\gamma$.

Example 3. If $\tau = \mathit{list}(T)$ we have for example that

$$\tau^{\langle\rangle} = \mathit{list}(T), \tau^{\langle([|],1)\rangle} = T \text{ and } \tau^{\langle([|],2)\rangle} = \tau^{\langle([|],2),([|],2)\rangle} = \mathit{list}(T).$$

Similarily for a term $t = [1, 2]$ we have for example that

$$t^{\langle\rangle} = [1, 2], t^{\langle([|],1)\rangle} = 1 \text{ and } t^{\langle([|],2)([|],1)\rangle} = 2.$$

In what follows, we will use the notion of a type-graph as a finite approximation of a possibly infinite type-tree. Therefore, we introduce the following equivalence relation on the paths in a type tree $\mathcal{L}_\tau$. We define $\equiv$ (in $\mathcal{L}_\tau$) as the least transitive relation such that for any $\delta, \alpha \in \mathcal{L}_\tau$: if $\delta = \alpha.\gamma$ and $\tau^\delta = \tau^\alpha$ then $\alpha \equiv \delta$. Informally, two type paths in a type tree are equivalent if either one of the paths is an extension of the other while both identify the same type, or the paths share a common initial subpath of the same type as both paths in $\mathcal{L}_\tau$. In what follows, we restrict our attention to (possibly polymorphic) types that are not defined in terms of a strict instance of itself. That is, we assume for any type τ and $\delta \in \mathcal{L}_\tau$ that there doesn't exist a type substitution θ such that $\tau^\delta = \tau\theta$. This is a natural condition and is related to the polymorphism discipline of definitional genericity [27]. For any such type τ, the equivalence relation $\equiv$ partitions the (possibly infinite set) $\mathcal{L}_\tau$ into a finite number of equivalence classes. For any $\delta \in \mathcal{L}_\tau$, the equivalence class of δ is defined as

$$[\delta] = \{\gamma \in \mathcal{L}_\tau \mid \delta \equiv \gamma\}.$$

The least element of an equivalence class $[\delta]$ exists and is defined as follows.

$$\overline{[\delta]} = \alpha \in [\delta] \text{ such that } \forall \beta \in [\delta] : \beta = \alpha.\gamma \text{ for some } \gamma \in \mathit{TPath}$$

Next, we define, for a type τ, its *type graph* as the finite set of minimal elements of the equivalence classes of $\mathcal{L}_\tau$:

Definition 4 (type-graph). *For a type $\tau \in \mathcal{T}(\Sigma_\mathcal{T}, V_\mathcal{T})$, we denote τ's type graph by $\mathcal{L}_\tau^\equiv$ which is defined as*

$$\mathcal{L}_\tau^\equiv = \{\overline{[\delta]} \mid \delta \in \mathcal{L}_\tau\}.$$

A type graph $\mathcal{L}_\tau^\equiv$ provides a finite approximation of the structure of terms of type τ: every path in $\mathcal{L}_\tau^\equiv$ abstracts a number of subterms of the term according to their type and position in the term. For the *list*(T) type from above, $\mathcal{L}_{list(T)}^\equiv = \{\langle\rangle, \langle([|], 1)\rangle\}$. The path $\langle\rangle$ represents all subterms of type *list*(T) in a term of type *list*(T), whereas $\langle([|], 1)\rangle$ represents all subterms of type T occurring in the first argument position of a functor $[|]$. In other words, $\langle\rangle$ can be seen as identifying the skeleton of the list, whereas $\langle([|], 1)\rangle$ as identifying the elements of the list. Note that as our notions of type-tree and type-graph describe the possible *positions* of subterms in terms of a particular type, they do not contain the zero-arity functors that possibly belong to the definition of the type. As such, our notions differ from more classic definions of type-trees and type-graphs like e.g. [22] or [39]. Also note that due to the particular definition of $\equiv$, two subterms of a same type are not necessarily abstracted by the same node in $\mathcal{L}_\tau^\equiv$. This is the case when $\mathcal{L}_\tau$ contains two type paths identifying the same type without them being equivalent, as in the next example.

Example 4. Consider the type *pair*(T) defined as

$$\mathit{pair}(T) \longrightarrow (T - T).$$

A term of the type $pair(T)$ is a term $(A - B)$ where A and B are terms of type T. For $\tau = pair(T)$,

$$typetree_\tau = \mathcal{L}_\tau^{\equiv} = \left\{ \begin{matrix} \langle\rangle \\ \langle(-),1\rangle \\ \langle(-),2\rangle \end{matrix} \right\}$$

Although $\langle(-),1\rangle$ and $\langle(-),2\rangle$ identify subterms of the same type T, they are not equivalent according to the definition of equivalence.

The ability to distinguish between two occurrences of the same type in $\mathcal{L}_\tau^{\equiv}$ allows a characterisation of terms of type τ with a finer granularity than with type based analyses [51, 7, 26]. This is illustrated with Example 4. A type based analysis places the two components of a pair in the same equivalence class as $\langle(-),1\rangle$ and $\langle(-),2\rangle$ select nodes of the same type. We do not and can calculate different binding times for them.

Now, one can obtain an abstract characterisation of terms of type τ, based on the structure of the term (or at least the type it belongs to), by associating an abstract value to each of the paths in $\mathcal{L}_\tau^{\equiv}$. For binding-time analysis, we are interested in the time a (part of a) value becomes known in the computation process. We use the abstract values $\mathcal{B} = \{static, dynamic\}$. *static* denotes that the binding certainly occurs at specialisation time; *dynamic* that it is not known when (and in case of logic programs "if") the binding occurs. A binding-time associates a value from $\mathcal{B}$ to each of the paths in a type graph.

Definition 5 (binding-time). *A* binding-time *for a type $t \in \mathcal{T}(\Sigma_{\mathcal{T}}, V_{\mathcal{T}})$ is a function*

$$\beta : \mathcal{L}_t^{\equiv} \mapsto \mathcal{B}$$

such that $\forall \delta \in dom(\beta)$ holds that $\beta(\delta) = dynamic$ implies that $\beta(\delta') = dynamic$ for all $\delta' \in dom(\beta)$ with $\delta' = \delta.\gamma$ for some $\gamma \in TPath$. The set of all binding-times (independent of the type) is denoted by $\mathcal{BT}$.

The relation between terms and the binding-times that approximate them is given by the following abstraction function.

Definition 6 (binding-time abstraction). *The* binding-time abstraction *is a function $\alpha : \mathcal{T}(\Sigma, \mathcal{V}) \mapsto \mathcal{BT}$ and is defined as follows:*

$$\alpha(t:\tau) = \left\{ (\delta, v) \left| \begin{array}{l} \delta \in \mathcal{L}_\tau^{\equiv} \textit{ and } v = dynamic \textit{ if } \exists\theta \textit{ and a subterm } t^{\delta'} \textit{ in } t\theta \\ \textit{such that } t^{\delta'} \textit{ is a variable and } \delta \equiv \delta' \\ v = static \textit{ otherwise} \end{array} \right. \right\}$$

If a term $t : \tau$ contains a subterm $t^{\delta'}$ that is a variable, then the binding-time abstraction associates the value *dynamic* to the path in $\mathcal{L}_\tau^{\equiv}$ that identifies this subterm and to all its extensions in $\mathcal{L}_\tau^{\equiv}$.

Example 5. Given the following terms of type $list(T)$ as defined in Example 1, their binding-time abstraction is:

$$\begin{array}{ll}\alpha([]) & = \{(\langle\rangle, static), (\langle([|], 1)\rangle, static)\} \\ \alpha([X_1, X_2] & = \{(\langle\rangle, static), (\langle([|], 1)\rangle, dynamic)\} \\ \alpha(X) & = \{(\langle\rangle, dynamic), (\langle([|], 1)\rangle, dynamic)\} \\ \alpha([X|Y]) & = \{(\langle\rangle, dynamic), (\langle([|], 1)\rangle, dynamic)\}\end{array}$$

Since the term [] does not contain any variable, it is abstracted by a binding-time specifying that the list's skeleton as well as its elements are *static*. A term $[X_1, X_2]$ is approximated by a binding-time specifying that the list's skeleton is *static*, but its elements are *dynamic*. A variable is abstracted by a binding-time specifying that the list's skeleton as well as its elements are *dynamic*. Also a term $[X|Y]$ is approximated by a binding-time stating that its list skeleton as well as its elements are dynamic due to the presence of the variable subterm $Y : list(T)$.

The following example shows why, if the value *dynamic* is associated to a path δ in a binding-time for a type τ, *dynamic* is also associated to all extensions of δ in $\mathcal{L}_\tau^{\equiv}$.

Example 6. Consider a type definition for a tree of integers:

```
inttree ---> nil ; t(int, inttree, inttree).
```

The type graph of $\tau = inttree$, $\mathcal{L}_\tau^{\equiv}$ contains only two paths: $\langle\rangle$ denoting the tree's skeleton, and $\langle t, 1\rangle$ denoting the integer elements in the tree. We have

$$\alpha(t(0, X, t(1, nil, nil))) = \{(\langle\rangle, dynamic), (\langle t, 1\rangle, dynamic)\}.$$

Although all subterms of type *int* in the term $t(0, X, t(1, nil, nil))$ are non-variable terms, we cannot abstract them to *static*. Indeed, the variable X in the term, being of type *inttree*, possibly represents some unknown integer elements.

To make our approximations suitable for a binding-time analysis, we define a partial order relation on $\mathcal{BT}$:

Definition 7 (covers). *Let β and $\beta' \in \mathcal{BT}$ such that $dom(\beta) \subseteq dom(\beta')$ or $dom(\beta') \subseteq dom(\beta)$. We say that β* covers *β', denoted by $\beta \succeq \beta'$ if and only if $\beta'(\delta) = dynamic \rightarrow \beta(\delta) = dynamic$ holds for all $\delta \in dom(\beta) \cap dom(\beta')$.*

If a binding-time β covers another binding-time β', then β is "at least as dynamic" as β'. Note that the relationship between $dom(\beta)$ and $dom(\beta')$ implies that the *covers* relation is only defined between two binding-times that are derived from types τ and τ' such that either τ is an instance of τ' or τ' is an instance of τ.

Example 7. Recall the binding-times obtained by abstracting the terms in Example 5. We have that

$$\alpha(X) \succeq \alpha([X_1, X_2]) \succeq \alpha([])$$

In what follows, we extend the notion of the $\succeq$ relation to include the elements $\{\top, \bot\}$ such that $\top \succeq \beta$ and $\beta \succeq \bot$ for all $\beta \in \mathcal{BT}$. If we denote with $\mathcal{BT}^+$ the set $\mathcal{BT}^+ = \mathcal{BT} \cup \{\top, \bot\}$, $(\mathcal{BT}^+, \succeq)$ forms a complete lattice. Wherever appropriate, we use $\bot$ and $\top$ to denote, for a particular type, a binding-time in which all paths are mapped to *static*, respectively a binding-time in which all paths are mapped to *dynamic*. Occasionally we will also call such binding-times completely static and completely dynamic, respectively.

We conclude this section by introducing some more notation. First, if β denotes a binding-time for a type τ and $\delta \in dom(\beta)$, then β^δ denotes the binding-time for a type τ^δ that is obtained as follows:

$$\beta^\delta = \left\{ (\gamma, \beta([\overline{\delta.\gamma}])) \middle| \gamma \in \mathcal{L}^{\equiv}_{\tau^\delta} \right\}.$$

In other words, if $\beta = \alpha(t)$ then $\beta^\delta = \alpha(t^\delta)$. Finally, let $\tau, \tau_1, \ldots, \tau_n$ be types and $f \in \Sigma$ such that $f(t_1 : \tau_1, \ldots, t_n : \tau_n)$ is a term in the denotation of τ. If $\beta_1, \ldots, \beta_n$ are binding-times for the types $\tau_1, \ldots, \tau_n$, we denote with $f(\beta_1, \ldots, \beta_n)$ the *least dynamic* binding-time for type τ such that $\beta^{[\overline{\langle (f,i) \rangle}]} \succeq \beta_i$ for all i.

3 A Modular Binding-time Analysis for Mercury

In what follows, we develop a polyvariant binding-time analysis. The final output of the analysis is an annotated program in which each of the original procedures may occur in several annotated versions, depending on the binding-times of the (input) arguments with respect to which the procedure was called. Each such version contains the binding-times of the local variables and output arguments as well as instructions stating for each subgoal of the procedure's body whether or not it should be evaluated during specialisation. Correctness of the analysis ensures that if a particular call $p(t_1, \ldots, t_n)$ occurs during specialisation, the analysis has created a version of the called procedure that is annotated with respect to the particular call's binding-time abstraction $p(\alpha(t_1), \ldots, \alpha(t_n))$. Before we define the actual analysis, we introduce Mercury's module system and define some necessary machinery to base the analysis upon.

3.1 Mercury's module system

A Mercury program is defined as a set of Mercury modules. The basic module system of Mercury is simple. A module consists of an *interface* part and an *implementation* part. The interface part contains those type definitions and procedure declarations that the module provides (or *exports*) towards other modules. In other words, the types and procedures declared in the interface part of a module are visible and can be used (or *imported*) by other modules. Apart from the implementation of the procedures that are declared in the module's interface, its implementation part possibly contains additional type definitions and the declaration and implementation of additional procedures. These types and procedures are only visible in the implementation part of this module, and can not be used by other modules.

Note that the way in which the modules import each other impose a hierarchy on the modules that constitute a program.[2] Following the terminology of [36], we use the notation *imports*(M, M') to indicate that the module M imports the interface of M' and *imported*(M) to denote the set of modules that are imported by M, that is: *imported*$(M) = \{M' \mid \textit{imports}(M, M')\}$. Figure 1 shows an example of a module hierarchy in Mercury in which we graphically represent a module by a box, and denote *imports*(M, M') by an arrow from M towards M'. In the example, we have that *imported*$(M_1) = \{M_2, M_3, M_5\}$. Note that in

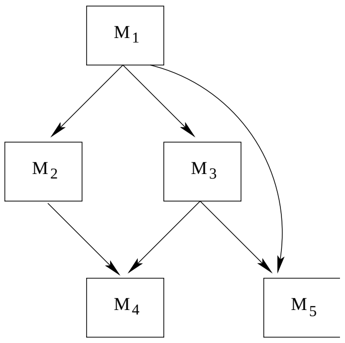

Fig. 1. A sample module hierarchy.

Mercury, the *imports* relation is not transitive; when a module M imports the interface of a module M', it becomes dependent on the interfaces imported by M' (and those imported therein) but it does not import these itself. The module system described above is to some extent a simplification of Mercury's real module system, in which modules can be constructed from submodules. While submodules do provide extra means to the programmer to control encapsulation and visibility of declarations, they do not pose additional conceptual difficulties and we do not consider them in the remainder of this work.

In this work, we aim at developing a binding-time analysis that is as modular as possible. Ultimately, a modular analysis deals with each module of a program in isolation. We will discuss throughout the text to what extent our binding-time analysis is modular in this respect.

3.2 Mercury programs for analysis

Mercury is an expressive language, in which programs can be composed of predicates and functions, one can use DCG notation, etc. However, if we consider only programs that are type correct and well-moded – which is natural, since the

[2] While in Mercury modules may depend on each other in a circular way, we restrict our attention to programs in which no circular dependencies exist between the modules. We discuss how one could deal with circular dependencies in Section 6.

compiler should reject programs that are not [43] – such a program can be translated into *superhomogeneous form* [43]. Translation to superhomogeneous form involves a number of analysis and transformation steps. These include translating an n-ary function definition into an $n+1$ ary predicate definition [44], making the implicit arguments in DCG-predicate definitions and calls explicit, and copying and renaming predicate definitions and calls such that every remaining predicate definition has a single mode declaration associated with it [43] that specifies for each argument whether it is an input or output argument. As such, every predicate definition is transformed to a set of so-called *procedure* definitions, with one procedure for every mode in which the original predicate is used.

For our analysis purposes, we assume that a Mercury program is given in superhomogeneous form. This does not involve any loss of generality, as the transformation from a plain Mercury program into superhomogeneous form is completely defined and automated [43]. Formally, the syntax of Mercury programs in superhomogeneous form can be defined as follows. We use the symbol Π to refer to the set of *procedure* symbols underlying the language associated to the program. As such, we consider two procedures that are derived from the same predicate as having different procedure symbols.

Definition 8 (superhomogeneous form).

$$\begin{array}{ll} \mathit{Proc} & ::= p(\overline{X}) :- G. \\ \mathit{Goal} & := \ \mathit{Atom} \mid \mathit{not}(G) \mid (G_1\ ,\ G_2) \mid (G_1\ ;\ G_2) \mid \mathit{if}\, G_1\ \mathit{then}\, G_2\ \mathit{else}\, G_3 \\ \mathit{Atom} & ::= X := Y \mid X == Y \mid X \Rightarrow f(\overline{Y}) \mid X \Leftarrow f(\overline{Y}) \mid p(\overline{X}) \end{array}$$

where $p/n \in \Pi$, X and Y are distinct variables and $\overline{X}$ is a sequence of n distinct variables of $\mathcal{V}$, $f/m \in \Sigma$, $\overline{Y}$ a sequence of m distinct variables of $\mathcal{V}$, and $G, G_1, G_2, G_3 \in \mathit{Goal}$.

The definition of a procedure p in superhomogeneous form consists of a single clause. The sequence of arguments in the head of the clause, denoted by $\mathcal{A}rgs(p)$, are distinct variables, explicit unifications are created for these variables in the body goal – denoted by $\mathcal{B}ody(p)$ – and complex unifications are broken down in several simpler ones. The arguments of a procedure p are divided in a set of input arguments, denoted by $\mathtt{in}(p)$ and a set of output arguments denoted by $\mathtt{out}(p)$. 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. Note that, as an effect of mode analysis [43], unifications are categorised as follows:

– An *assignment* of the form $X := Y$. For such a unification, Y is input, whereas X is output.
– A *test* of the form $X == Y$. Both X and Y are input to the unification and of atomic type.
– A *deconstruction* of the form $X \Rightarrow f(\overline{Y})$. In this case, X is input to the unification whereas $\overline{Y}$ is a sequence of output variables.

– A *construction* of the form $X \Leftarrow f(\overline{Y})$. In this case X is output from the unification whereas $\overline{Y}$ is a sequence of input variables.

During the translation into superhomogeneous form, unifications between values of a complex data type may be transformed into a call to a newly generated procedure that (possibly recursively) performs the unification. For any goal G, we denote with $\texttt{in}(G)$ and $\texttt{out}(G)$ the set of its input, respectively output variables[3]

Example 8. Consider the classical definition of the `append/3` predicate, both in normal syntax and in superhomogeneous form for the mode `append(in,in,out)` as depicted in Fig. 2.

`append/3`	`append/3` in superhomogeneous form
append([],Y,Y).	append(X,Y,Z):-
append([E|Es],Y,[E|R]):-	(X⇒[], Z:=Y ;
append(Xs,Y,R).	X⇒[E|Es], append(Es, Y, R), Z⇐[E|R]).

Fig. 2. The `append/3` predicate and `append(in,in,out)` in superhomogeneous form.

According to Definition 8, conjunctions and disjunctions are considered binary constructs. This differs from their representation inside the Melbourne compiler [40], where conjunctions and disjunctions are represented in flattened form. Our syntactic definition however facilitates the conceptual handling of these constructs during analysis.

For analysis purposes, we assume that every subgoal of a procedure body is identified by a unique program point, the set of all such program points is denoted by $\mathcal{P}p$. If we are dealing with a particular procedure, we denote with η_0 the program point associated with the procedure's head atom, and with η_b the program point associated to its body goal. The set of program points identifying the subgoals of a goal G is denoted by $\mathcal{P}ps(G)$, this set includes the program point identifying G itself. If the particular program point identifying a goal G in a procedure's body is important, we subscribe the goal with its program point, as in G_η or explicitly state that $\mathcal{P}p(G) = \eta$. An important use of program points is to identify those atoms in the body of a procedure in which a particular variable becomes initialised or, said otherwise, those atoms of which the variable is an output variable. This information is computed by mode analysis, and we assume the availability of a function

$$\texttt{init} : \mathcal{V} \mapsto \wp(\mathcal{P}p)$$

[3] Although Mercury has some support for more involved modes – other than input versus output – that are necessary to support *partially instantiated data structures* at run-time, release 0.9 of the Mercury implementation [40] does not fully support these.

with the intended meaning that, for a variable V used in some procedure, if $\texttt{init}(V) = \{\eta_1, \ldots, \eta_n\}$, the variable V is an output variable of the atoms identified by $\eta_1, \ldots, \eta_n$. Note that the function `init` is implicitly associated with a particular procedure, which we do not mention explicitly. When we use the function `init`, it will be clear from the context to what particular procedure it is associated.

Example 9. Let us recall the definition of `append/3` in superhomogeneous form for the mode `append(in,in,out)`, with the atoms and structured goals occurring in the procedure's definition explicitly identified by subscribing them with their respective program point as in Figure 3. We denote the program points

```
append(X,Y,Z)_0:-
  ((X⇒[]_1, Z:=Y_2)_c1 ;
   (X⇒[E|Es]_3, (append(Es, Y, R)_4, Z⇐[E|R]_5)_c2)_c3)_d1.
```

Fig. 3. `append/3` with explicit program points.

associated to a structured goal by subscripting the goal with the characters 'c' for conjunction and 'd' for disjunction, accompanied by a natural number. From mode analysis, it follows that

$$\begin{array}{lll} \texttt{init}(X) = \{0\} & \texttt{init}(E) = \{3\} & \texttt{init}(R) = \{4\} \\ \texttt{init}(Y) = \{0\} & \texttt{init}(Es) = \{3\} & \texttt{init}(Z) = \{2, 5\} \end{array}$$

Or, put otherwise, X and Y (being input arguments) are initialised in the procedure's head, E and Es are initialised in the deconstruction identified by program point 3, R is initialised in the recursive call whereas Z is initialised either by the assignment $Z := Y$ (program point 2) or by the construction $Z \Leftarrow [E|R]$ (program point 5).

3.3 A modular analysis

In order to make the binding-time analysis as modular as possible, we devise an analysis that works in two phases. In a first phase, we represent binding-times and the relations that exist between them according the data flow in the program in a symbolic way. Doing so enables us to perform a large part of the data-flow analysis independent of a particular call pattern. It is only in the second phase that call patterns in the form of the binding-times of a procedure's input arguments are used — in combination with the symbolic information derived from the first phase— for computing the annotations and the actual binding-times of the procedure's other variables. The first phase of the analysis hence is *call independent* whereas the second phase is *call dependent.* Obviously, the call independent phase of the analysis does not need to be repeated in case a procedure is called with a different binding-time characterisation of its arguments and consequently, the result of a module's call independent analysis can be used regardless of the context the module is used in, and must not be repeated when the module is used in different programs. Since the domain of binding-times is condensing [21], the call-independent analysis preserves the precision that would be obtained by a call-dependent analysis.

To symbolically represent the binding-time of a variable at a particular program point, we introduce the concept of a *binding-time variable*, the set of which is denoted by $\mathcal{V}_{\mathcal{BT}}$. We will denote elements of this set as variables subscribed by a program point. If V is a variable occurring in a goal G, and η is a program point identifying an atom in G, then the binding-time variable $V_\eta \in \mathcal{V}_{\mathcal{BT}}$ symbolically represents the binding-time of V at program point η. Given a type path $\delta \in \mathit{TPath}$, we use the notation V_η^δ to denote the subvalue identified by δ in the binding-time of V at program point η[4].

Example 10. Given the definition of `append/3` from Example 9, the binding-time variables X_0, Z_2, Z_5 and Z_0 denote, respectively the binding-time of X at the program point 0 and the binding-times of Z at the program points $2, 5$ and 0.

Apart from the binding-time variables that correspond with program variables, we introduce a number of extra binding-time variables that we use to symbolically represent some control information that will be collected (and needed) during the binding-time analysis. For each program point η, we introduce two such variables, $\mathcal{R}_\eta$ and $\mathcal{C}_\eta$, that range over the set of binding-times $\{\bot, \top\}$. Their intended meaning is as follows:

– $\mathcal{R}_\eta = \bot$: Either the goal identified by η reduces to *true* or *fail* during specialisation, or its residual code is guaranteed not to fail at run-time.
– $\mathcal{R}_\eta = \top$: No claims are made about the outcome of the reduction at specialisation time.
– $\mathcal{C}_\eta = \top$: The goal identified by η is under dynamic control in the procedure's body. We say that an atom is under dynamic control if the fact whether it will be evaluated depends on the success or failure of another goal, say $G_{\eta'}$ while success or failure of that goal is undecided at specialisation-time (that is $\mathcal{R}_{\eta'} = \top$).
– $\mathcal{C}_\eta = \bot$: The goal identified by η is not under dynamic control in the procedure's body.

Note that these binding-time variables – which we will refer to as *control variables* – are boolean in the sense that they will only assume a value that is either $\bot$ or $\top$. During the binding-time analysis, these control variables collect the necessary information to implement the control strategy of the specialiser. Our analysis models a rather conservative specialisation strategy, in the sense that during specialisation, no atoms are reduced that are under dynamic control.

[4] Hence $V_\eta^{\langle\rangle}$ and V_η denote the same binding-time value and we will use the latter in examples.

The idea behind this strategy is that in this way only atoms are reduced that would also be evaluated if the program is executed with a complete input that extends the static input for which the program is specialised.

Indeed, their being evaluated depends only on goals that are – during specialisation– sufficiently reduced in order to decide success or failure. Hence, no atoms are "speculatively" reduced, guaranteeing termination of the reduction process (constituting local termination) under the assumption that the equivalent single stage computation terminates.

Example 11. Consider the following code fragment

```
if X ⇒ [] then p(X) else q(X)
```

Both atoms $p(X)$ and $q(X)$ are under dynamic control if X's binding-time does not allow the specialiser to decide whether or not the test $X \Rightarrow []$ will succeed during specialisation. Indeed, the specialiser has no means of knowing which of the branches will be taken during the second stage of the computation.[5]

In general, the binding-time of a program variable can depend on the binding-times of other program variables (according to the data flow) and on the value of the appropriate control variables (according to the control strategy). The values of the control variables that are associated to a goal in turn depend on the binding-times of that goal's input variables. Symbolically, we can represent these dependencies by a number of constraints between the involved binding-time variables. In general:

Definition 9 (binding-time constraint). *A* binding-time constraint *is a constraint of the following form:*

$$V_\eta^\delta \succeq X_{\eta'}^\gamma,\ V_\eta^\delta \succeq \top$$

$$V_\eta^\delta \succeq^* X_{\eta'}^\gamma,\ V_\eta^\delta \succeq^* \top$$

where $V_\eta, X_{\eta'} \in \mathcal{V}_{\mathcal{BT}}$ and $\delta, \gamma \in TPath$. The set of all binding-time constraints is denoted by $\mathcal{BTC}$.

A constraint of the form $V_\eta^\delta \succeq X_{\eta'}^\gamma$ denotes that the binding-time represented by V_η^δ must be at least as dynamic as (or *cover*) the binding-time represented by $X_{\eta'}^\gamma$. Note that such a constraint requires the types of V and X, denoted by τ_V and τ_X to be such that τ_V^δ and τ_X^γ are instances of one another, in order

[5] Note that it can happen that the analysis cannot predict the outcome of the test while execution of the program with full input always selects the same branch, e.g. $q(X)$. Although the call to $p(X)$ is residualised, the code of the procedure $p/1$ is specialised. All reductions performed while specialising $p/1$ are then in fact speculative (and the specialisation could in extreme cases be non-terminating while execution of the program to be specialised with full input is always terminating).

for their binding-times to be comparable. The intended meaning of a constraint of the form $V_\eta^\delta \succeq^* X_{\eta'}^\gamma$ is that the binding-time represented by V_η^δ is at least as dynamic as the binding-time value associated to the path identified by γ in the binding-time represented by $X_{\eta'}^\gamma$. Note that such a constraint does not require τ_V^δ and τ_X^γ to be of comparable types; it simply expresses that if the node identified by γ in the binding-time represented by $X_{\eta'}$ is *dynamic*, so must be the node identified by δ in V_η and by definition of a binding-time, so must be all its descendant nodes. Remark that we also allow constraints in which the right-hand side is the constant $\top$. Although we occasionally also consider constraints of which the right-hand side is the constant $\bot$, we do not explicitly mention these in the definition, as these constraints are superfluous: for any $X_\eta \in \mathcal{V}_{\mathcal{BT}}$ and $\delta \in \mathit{TPath}$, it holds by definition that $X_\eta^\delta \succeq \bot$.

Example 12. Reconsider the definition of `append/3` in Fig. 3. Some examples of binding-time constraints between binding-time variables from `append/3` and their intended meaning are:

$Z_2 \succeq Y_0$	The binding-time associated to Z at program point 2 is at least as dynamic as the binding-time associated to Y at program point 0.
$E_3 \succeq X_0^{\langle [\|],1\rangle}$	The binding-time associated to E at program point 3 is at least as dynamic as the subvalue denoted by $\langle [\|],1\rangle$ of the binding-time associated to X at program point 0.
$Z_5^{\langle [\|],1\rangle} \succeq E_3$	The subvalue denoted by $\langle [\|],1\rangle$ in the binding-time of Z at program point 5 is at least as dynamic as the binding-time associated to E at program point 3.
$\mathcal{R}_3 \succeq^* X_0$	If X_0 represents a binding-time in which the root node $\langle\rangle$ is bound to *dynamic* then one cannot assume that the atom at program point 3 reduces to *true*, *fail* or code that is guaranteed to succeed.
$\mathcal{C}_4 \succeq \mathcal{R}_3$	The atom at program point 4 must be under dynamic control if the specialisation of the atom at program point 3 possibly results in residual code that might fail at run-time.

A set of binding-time constraints is called a binding-time constraint system (or simply a constraint system). Given a constraint system $\mathcal{C}$, we define vars($\mathcal{C}$) as the set of all binding-time variables X_η that occur in some constraint $C \in \mathcal{C}$. The link between a binding-time constraint system and the actual binding-times it represents is formalised as a (minimal) solution to the constraint system.

Definition 10 (solution). *A* solution *to a binding-time constraint system $\mathcal{C}$ is a substitution $\sigma : \mathcal{V}_{\mathcal{BT}} \mapsto \mathcal{BT}$ mapping binding-time variables to binding-times with $dom(\sigma) = vars(\mathcal{C})$ such that*

- *for every constraint $V_\eta^\delta \succeq \top \in \mathcal{C}$ and $V_\eta^\delta \succeq^* \top \in \mathcal{C}$ it holds that $\sigma(V_\eta)^\delta \succeq \top$*
- *for every constraint $V_\eta^\delta \succeq X_{\eta'}^\gamma \in \mathcal{C}$ it holds that $\sigma(V_\eta)^\delta \succeq \sigma(X_{\eta'})^\gamma$*
- *for every constraint $V_\eta^\delta \succeq^* X_{\eta'}^\gamma \in \mathcal{C}$ it holds that $\sigma(X_{\eta'})(\gamma) = dynamic \Rightarrow \sigma(V_\eta)^\delta \succeq \top$*

Given two solutions σ and σ' to $\mathcal{C}$, we define that $\sigma \sqsupseteq \sigma'$ if for all $V_\eta \in dom(\sigma')$ it holds that $V_\eta \in dom(\sigma)$ and $\sigma(V_\eta) \succeq \sigma'(V_\eta)$. A solution σ is a least solution *for $\mathcal{C}$ if for every solution σ' for $\mathcal{C}$ it holds that $\sigma' \sqsupseteq \sigma$.*

Remember, a solution must also satisfy the condition of Definition 5, i.e. if $\sigma(X_{\eta'})^\gamma = \mathit{dynamic}$ then also $\sigma(X_{\eta'})^{\gamma\cdot\alpha} = \mathit{dynamic}$ for any extension α. We will sometimes use a constraint of the form $V_\eta^\delta \succeq X_{\eta'}^{\gamma'} \sqcup Y_{\eta''}^{\gamma''}$ (analogously for $\succeq^*$) as shorthand notation for the set of constraints $\{V_\eta^\delta \succeq X_{\eta'}^{\gamma'}, V_\eta^\delta \succeq Y_{\eta''}^{\gamma''}\}$. Indeed, from Definition 10 it can be seen that in any solution σ satisfying the latter two constraints, it holds that $\sigma(V_\eta)^\delta \succeq \sigma(X_{\eta'}^{\gamma'}) \sqcup \sigma(Y_{\eta''}^{\gamma''})$, where $\sqcup$ denotes the least upper bound on $(\mathcal{BT}^+, \succeq)$.

Example 13. Consider the following binding-time constraint system and its least solution. For sake of simplicity, we assume that all binding-time variables are boolean and range over the set $\{\mathit{dynamic}, \mathit{static}\}$.

Binding-time constraint system	Least solution
$X_{\eta_1} \succeq \top$ $R_{\eta_3} \succeq X_{\eta_2}$ $Y_{\eta_4} \succeq X_{\eta_1}$ $Y_{\eta_4} \succeq R_{\eta_3}$	$\left\{\begin{array}{ll} (X_{\eta_1}, \mathit{dynamic}) & (X_{\eta_2}, \mathit{static}) \\ (R_{\eta_3}, \mathit{static}) & (Y_{\eta_4}, \mathit{dynamic}) \end{array}\right\}$

In what follows, we formulate our analysis as a call-independent abstract semantics. We define the abstract "meaning" of a goal, be it an atom or a structured goal, as a set of binding-time constraints (description domain $\wp(\mathcal{BTC})$) that reflect the data flow between the input- and output arguments of the goal. An essential operator for the symbolic data flow analysis is a projection operator that basically rewrites a set of constraints such that every constraint expresses (or constrains) the binding-time of a local variable within a procedure in function of the binding-time(s) of that procedure's input arguments. Such a constraint is said to be in normal form:

Definition 11 (normal form). *A binding-time constraint is in* normal form *with respect to a procedure $p \in Proc$ if it is either of the form*

- $V_\eta^\delta \succeq \top$
- $V_\eta^\delta \succeq X_{\eta_0}^\gamma$ *with $X \in$ `in`(p) and η_0 the program point associated to p's head atom.*

and analogously for constraints of this form using $\succeq^*$.

Example 14. Reconsider the binding-time constraints from Example 12. The constraints

$$Z_2 \succeq Y_0 \qquad E_3 \succeq X_0^{\langle [|],1\rangle} \qquad \mathcal{R}_3 \succeq^* X_0$$

are in normal form with respect to `append/3`, whereas the constraints

$$Z_5^{\langle [|],1\rangle} \succeq E_3 \qquad \mathcal{C}_4 \succeq \mathcal{R}_3$$

are not.

Projection of a constraint involves unfolding the (subvalue of the) binding-time variable in its right-hand side with respect to a single constraint on (a subvalue of) this variable. If we consider two subvalues of a binding-time variable, say X_η^δ and X_η^γ, one of them is a subvalue of the other if either δ is an extension of γ or vice versa. This is captured by the following definition:

Definition 12 (extension). *We define* $ext : TPath \times TPath \mapsto TPath \times TPath$ *as follows:*

$$ext(\gamma, \delta) = \begin{cases} (\langle\rangle, \alpha) & \textit{if } \gamma = \delta.\alpha \\ (\alpha, \langle\rangle) & \textit{if } \gamma.\alpha = \delta \\ \textit{undefined otherwise} \end{cases}$$

Note that if $\text{ext}(\gamma, \delta) = (\alpha, \alpha')$ then $\gamma.\alpha = \delta.\alpha'$. Unfolding a constraint $X_\eta^\gamma \succeq Y_{\eta'}^\delta$ with respect to another constraint result in a new constraint on (a subvalue of) X_η^γ, with as right hand side the appropriate subvalue of the right hand side of the constraint that was used for unfolding. To denote a subvalue of a constraint's right hand side ϕ (which is either a binding-time variable or one of the constants $\top$ or $\bot$), we use the notation $\phi^{\overline{\cdot\alpha}}$. If ϕ denotes a variable X_η^γ, then $\phi^{\overline{\cdot\alpha}}$ equals $X_\eta^{\overline{[\gamma.\alpha]}}$. Otherwise, if ϕ denotes one of the constants $\bot$ or $\top$, $\phi^{\overline{\cdot\alpha}}$ simply equals ϕ. Note the use of the least element of the equivalence class, $\overline{[\gamma.\eta]}$, to denote an element of the appropriate type graph $\mathcal{L}_\tau^{\equiv}$ (rather than the type tree $\mathcal{L}_\tau$). The projection operation is defined in Definition 13 and basically consists of a fixed point iteration over an unfolding operator followed by a selection operation that retrieves the constraints of interest from the fixed point. Recall that η_0 identifies the head atom of the procedure of interest.

Definition 13 (projection). *The* projection *of a set* $S \subseteq \wp(\mathcal{BTC})$ *on a set of binding-time variables* $V \subseteq \mathcal{V}_{\mathcal{BT}}$ *is denoted by* $proj_V S$ *and defined as*

$$proj_V(S) = \{X \succeq^{(*)} \phi \in lfp(unf_S) \mid X \in V\}$$

where unf_S *is defined in Figure 4.*

The symbolic analysis is defined in Definition 14. The result of analysing a program is a mapping (from the semantic domain *Den*) that maps a procedure symbol p to a set of binding-time constraints on the variables that occur in the definition of the procedure p. The constraints are in normal form. Polyvariance is immediate, since all constraints are expressed in terms of the procedure's input arguments, which are represented symbolically and hence can be instantiated by any call pattern. The analysis is defined by a number of semantic functions defining the abstract semantics of a program $\mathbf{P} : Prog \mapsto Den$ in terms of the semantics of the individual procedures, goals and atoms.

Definition 14 (call independent abstract semantics). *The* call independent abstract semantics *for description domain* $\wp(\mathcal{BTC})$ *has semantic domain*

$$Den : \Pi \mapsto \wp(\mathcal{BTC})$$

$$\text{unf}_S : \wp(\mathcal{BTC}) \mapsto \wp(\mathcal{BTC})$$

$$\text{unf}_S(I) = \left\{ C \;\middle|\; \begin{array}{l} C \in S \cup S_1 \cup S_2 \cup S_3 \text{ and the form of } C \text{ is} \\ \text{either} X_\eta \succeq^{(*)} Y_{\eta_0} \text{ or } X_\eta \succeq^{(*)} \top \end{array} \right\}$$

where

$$S_1 = \{X_\eta^{\overline{[\gamma \cdot \alpha]}} \succeq \phi^{\overline{\cdot \alpha'}} \mid X_\eta^\gamma \succeq Y_{\eta'}^\delta \in S, Y_{\eta'}^{\delta'} \succeq \phi \in I, \text{ and } \text{ext}(\delta, \delta') = (\alpha, \alpha')\}$$

$$S_2 = \{X_\eta^\gamma \succeq^* \phi \mid X_\eta^\gamma \succeq Y_{\eta'} \in S \text{ and } Y_{\eta'} \succeq^* \phi \in I\}$$

$$S_3 = \{X_\eta \succeq^* \phi^{\overline{\cdot \alpha}} \mid X_\eta \succeq^* Y_{\eta'}^\delta \in S, Y_{\eta'}^{\delta'} \succeq \phi \in I \text{ and } \text{ext}(\delta, \delta') = (\langle\rangle, \alpha)\}$$

Fig. 4. The projection proj_V

and semantic functions

$$\mathbf{P} : \mathit{Prog} \mapsto \mathit{Den}$$
$$\mathbf{C} : \mathit{Proc} \mapsto \mathit{Den} \mapsto \mathit{Den}$$
$$\mathbf{G} : \mathit{Goal} \mapsto \mathit{Den} \mapsto \wp(\mathcal{BTC})$$
$$\mathbf{A} : \mathit{Atom} \mapsto \mathit{Den} \mapsto \wp(\mathcal{BTC})$$

and is defined in Figures 5 and 6.

The result of analysing a program is a denotation, $\mathbf{P}[\![P]\!]$, in the domain Den, which is a mapping from a predicate symbol to a set of binding-time constraints. This mapping is defined as the least fixed point of applying the analysis function $\mathbf{C}$ to each individual procedure. The analysis function $\mathbf{C}$ constructs a partial denotation for a particular procedure, given a (possibly incomplete) denotation that represents the result of analysis of the whole program so far. The analysis functions $\mathbf{G}$ and $\mathbf{A}$ map respectively a structured goal and an atomic goal to a set of binding-time constraints, given a denotation – again representing the result of analysing the whole program so far. In general, the result of analysing a complex goal is the union of the constraints obtained by analysing each subgoal in isolation, together with a number of additional constraints on the control variables associated with the goal and its subgoals. These constraints are simple, as they merely reflect the propagation of the control variable's value, either from the goal to its subgoals (in case of the control variable $\mathcal{C}$) or from the goal's subgoals to the goal itself (in case of $\mathcal{R}$). The binding-time variables denoting dynamic control denote that a goal is under dynamic control *with respect to the procedure's body.* The negated goal (G) in a negation is under dynamic control only if the negation ($\neg G$) itself is. Observe that if A reduces to true or is guaranteed to succeed, then not(A) fails. And if A fails then not(A) succeeds. So we can say that the negation reduces to true, fail, or residual code which is guaranteed to succeed if the negated goal does. The propagation in the other

$$\mathbf{P}[\![P]\!] = \text{lfp}(\bigcup_{p \in Proc(P)} \mathbf{C}[\![p]\!])$$

$$\mathbf{C}[\![p(\overline{X}) \leftarrow G_\eta]\!]d = \{(p, \mathbf{G}[\![G_\eta]\!]d)\}$$

$$\begin{array}{ll}
\mathbf{G}[\![(G'_{\eta'}, G''_{\eta''})_\eta]\!]d & = \mathbf{G}[\![G'_{\eta'}]\!]d \cup \mathbf{G}[\![G''_{\eta''}]\!]d \cup CC_{\text{conj}}(\eta, \eta', \eta'') \\
\mathbf{G}[\![not_\eta(G_{\eta'})]\!]d & = \mathbf{G}[\![G_{\eta'}]\!]d \cup CC_{\text{not}}(\eta, \eta') \\
\mathbf{G}[\![if_\eta\ G'_{\eta'}\ then\ G''_{\eta''}\ else\ G'''_{\eta'''}]\!] & = \mathbf{G}[\![G'_{\eta'}]\!]d \cup \mathbf{G}[\![G''_{\eta''}]\!]d \cup \mathbf{G}[\![G'''_{\eta'''}]\!]d \\
& \quad \cup CC_{\text{if}}(\eta, \eta', \eta'', \eta''') \\
\mathbf{G}[\![(G'_{\eta'}; G''_{\eta''})_\eta]\!]d & = \mathbf{G}[\![G'_{\eta'}]\!]d \cup \mathbf{G}[\![G''_{\eta''}]\!]d \cup CC_{\text{disj}}(\eta, \eta', \eta'') \\
\mathbf{G}[\![A_\eta]\!]d & = \mathbf{A}[\![A_\eta]\!]d \\
& \quad \cup \{X_\eta \succeq X_{\eta'} \mid X \in \texttt{in}(A), \eta' \in \texttt{reach}(X, \eta)\}
\end{array}$$

$$\begin{array}{ll}
\mathbf{A}[\![X ==_\eta Y]\!]d & = \{\mathcal{R}_\eta \succeq^* X_\eta \sqcup Y_\eta\} \\
\mathbf{A}[\![X :=_\eta Y]\!]d & = \{X_\eta \succeq Y_\eta \sqcup \mathcal{C}_\eta,\ \mathcal{R}_\eta \succeq \bot\} \\
\mathbf{A}[\![X \Rightarrow_\eta f(\overline{Y})]\!]d & = \bigcup_{Y_i \in \overline{Y}} \{Y_{i_\eta} \succeq X_\eta^{\overline{[\langle f,i \rangle]}} \sqcup \mathcal{C}_\eta\} \cup \{\mathcal{R}_\eta \succeq^* X_\eta\} \\
\mathbf{A}[\![X \Leftarrow_\eta f(\overline{Y})]\!]d & = \bigcup_{Y_i \in \overline{Y}} \{X_\eta^{\overline{[\langle f,i \rangle]}} \succeq Y_{i_\eta} \sqcup \mathcal{C}_\eta\} \cup \{\mathcal{R}_\eta \succeq \bot\} \\
\mathbf{A}[\![p(X_1, \ldots, X_n)_\eta]\!]d & = \rho(\text{proj}_{\mathcal{A}rgs(p), \mathcal{R}_{\eta_b}}\, d\, p) \sqcup \{X_{i_\eta} \succeq \mathcal{C}_\eta \mid X_i \in \texttt{out}(p)\}
\end{array}$$

where $\mathcal{A}rgs(p)$ denotes the sequence of formal arguments in the definition of p/n, η_b is associated to the body goal in the definition of p/n and ρ is a renaming mapping the sequence of formal arguments $\mathcal{A}rgs(p)$ to the sequence of actual arguments $\langle X_1, \ldots, X_n \rangle$ and $\mathcal{R}_{\eta_b}$ to $\mathcal{R}_\eta$.

Fig. 5. The call independent abstract semantics

$$CC_{\text{conj}}(\eta, \eta', \eta'') = \left\{ \begin{array}{c} \mathcal{C}_{\eta'} \succeq \mathcal{C}_\eta\ \mathcal{C}_{\eta''} \succeq \mathcal{C}_\eta\ \mathcal{C}_{\eta''} \succeq \mathcal{R}_{\eta'} \\ \\ \mathcal{R}_\eta \succeq \mathcal{R}_{\eta'}\ \mathcal{R}_\eta \succeq \mathcal{R}_{\eta''} \end{array} \right\}$$

$$CC_{\text{disj}}(\eta, \eta', \eta'') = \left\{ \begin{array}{c} \mathcal{C}_{\eta'} \succeq \mathcal{C}_\eta\ \mathcal{C}_{\eta''} \succeq \mathcal{C}_\eta \\ \\ \mathcal{R}_\eta \succeq \mathcal{R}_{\eta'}\ \mathcal{R}_\eta \succeq \mathcal{R}_{\eta''} \end{array} \right\}$$

$$CC_{\text{not}}(\eta, \eta') = \left\{ \mathcal{C}_{\eta'} \succeq \mathcal{C}_\eta\ \mathcal{R}_\eta \succeq \mathcal{R}_{\eta'} \right\}$$

$$CC_{\text{if}}(\eta, \eta', \eta'', \eta''') = \left\{ \begin{array}{c} \mathcal{C}_{\eta'} \succeq \mathcal{C}_\eta\ \mathcal{C}_{\eta''} \succeq \mathcal{C}_\eta\ \mathcal{C}_{\eta'''} \succeq \mathcal{C}_\eta \\ \\ \mathcal{C}_{\eta''} \succeq \mathcal{R}_{\eta'}\ \mathcal{C}_{\eta'''} \succeq \mathcal{R}_{\eta'} \\ \\ \mathcal{R}_\eta \succeq \mathcal{R}_{\eta'}\ \mathcal{R}_\eta \succeq \mathcal{R}_{\eta''}\ \mathcal{R}_\eta \succeq \mathcal{R}_{\eta'''} \end{array} \right\}$$

Fig. 6. The call independent abstract semantics (ctd.)

constructs is similar: the subgoals of an if-then-else are under dynamic control if the if-then-else is under dynamic control. Moreover, both the then and else goals are under dynamic control if the test goal possibly reduces to residual code which could fail at run time. If each of the if-then-else's subgoals reduces to true, fail or code that is guaranteed to succeed, so does the if-then-else. The subgoals of a conjunction are under dynamic control if the conjunction itself is. Moreover, the second conjunct is under dynamic control if the first conjunct possibly reduces to residual code that could fail. If both conjuncts reduce to true, fail or code that is guaranteed to succeed, so does the conjunction. To conclude, if a disjunction is under dynamic control, so are both disjuncts. If both disjuncts reduce to true, fail or code that is guaranteed to succeed, so does the disjunction.

Example 15. Reconsider the definition of `append/3` in Figure 3. The body goal contains the following structured subgoals: a conjunction identified by program point c_1 with the atomic conjuncts identified by program points 1 and 2, a second conjunction identified by c_2 with the atomic conjuncts identified by program points 4 and 5, a third conjunction identified by c_3 with the conjuncts identified by program points 3 and c_2 and a disjunction identified by program point d_1 with the disjuncts identified by c_1 and c_3. The binding-time constraints that are associated to each of these structured goals are as follows:

(c_1)	$\mathcal{C}_1 \succeq \mathcal{C}_{c_1}$	$\mathcal{R}_{c_1} \succeq \mathcal{R}_1$
	$\mathcal{C}_2 \succeq \mathcal{C}_{c_1}$	$\mathcal{R}_{c_1} \succeq \mathcal{R}_2$
	$\mathcal{C}_2 \succeq \mathcal{R}_1$	
(c_2)	$\mathcal{C}_4 \succeq \mathcal{C}_{c_2}$	$\mathcal{R}_{c_2} \succeq \mathcal{R}_4$
	$\mathcal{C}_5 \succeq \mathcal{C}_{c_2}$	$\mathcal{R}_{c_2} \succeq \mathcal{R}_5$
	$\mathcal{C}_5 \succeq \mathcal{R}_4$	
(c_3)	$\mathcal{C}_3 \succeq \mathcal{C}_{c_3}$	$\mathcal{R}_{c_3} \succeq \mathcal{R}_3$
	$\mathcal{C}_{c_2} \succeq \mathcal{C}_{c_3}$	$\mathcal{R}_{c_3} \succeq \mathcal{R}_{c_2}$
	$\mathcal{C}_{c_2} \succeq \mathcal{R}_3$	
(d_1)	$\mathcal{C}_{c_1} \succeq \mathcal{C}_{d_1}$	$\mathcal{R}_{d_1} \succeq \mathcal{R}_{c_1}$
	$\mathcal{C}_{c_3} \succeq \mathcal{C}_{d_1}$	$\mathcal{R}_{d_1} \succeq \mathcal{R}_{c_3}$

The binding-time constraints that are associated to an atomic goal are somewhat more involved. Apart from binding-time constraints on the atom's output variables, analysing an atom A_η also possibly results in a binding-time constraint on the control variable $\mathcal{R}_\eta$, indicating under what conditions the atom can be reduced to true, fail, or code that is guaranteed to succeed. Moreover, when creating the binding-time constraints on the atom's output variables, the control variable $\mathcal{C}_\eta$ must be taken into account, in order to guarantee that the particular binding-time is made $\top$ in case the atom is under dynamic control.

Note that in the definition of **A** the binding-time variables that refer to the *input* variables of an atom at program point η are indexed by the program point η. Consequently, a number of additional constraints must be created for each atom, relating the binding-time of such an input argument at program point η with its binding-time at the program point(s) where the binding-time was created, being output of some other atom.

A test does not have any output variables, so it only creates constraints on control variables. The atom reduces to true, fail or code that is guaranteed to succeed when both input variables are bound to an outermost functor. An assignment $X := Y$ introduces the constraints specifying that the binding-time of X at program point η must be at least as dynamic as the binding-time of Y at program point η. Recall that the latter's value is constrained to be at least as dynamic as the least upper bound of the binding-times of Y at the reachable program points where Y is assigned a value. Moreover, if the assignment is under dynamic control, X_η must be assigned the value $\top$. This is guaranteed by adding $\sqcup\, \mathcal{C}_\eta$ to the right-hand side of the constraint on X_η. Even if an assignment is not reduced, it can never fail at run time. Hence the (superfluous) constraint $\mathcal{R}_\eta \succeq \bot$. A deconstruction introduces some binding-time constraints indicating that the binding-time of the newly introduced variables must be at least as dynamic as the corresponding subvalue in the binding-time of the variable that is deconstructed. Also in this case, the least upper bound with $\mathcal{C}_\eta$ guarantees that, if the deconstruction is under dynamic control, the newly introduced binding-time variables will be forced to have the value $\top$. If the deconstructed variable is bound to at least an outermost functor, the deconstruction reduces to true or fail at specialisation time. Otherwise, a residualised deconstruction can either succeed or fail at run time which is reflected by the fact that in that case $\mathcal{R}_\eta$ will have the value $\top$. When handling a construction on the other hand, the binding-time of the constructed variable is constrained by the binding-times of the variables used in the construction. Again, if the construction is under dynamic control, the constructed binding-time is guaranteed to be $\top$ by the use of the least upper bound with $\mathcal{C}_\eta$. Even when residualised, a construction can never fail, so again the (superfluous) constraint $\mathcal{R}_\eta \succeq \bot$ is introduced.

Example 16. Reconsider the definition of `append/3` in Figure 3. The constraints that are associated to the unifications in `append/3`'s body goal are as follows. The numbers in the left hand side column denote the particular unification's program point.

(1)	$\mathcal{R}_1 \succeq^* X_0$	
(2)	$\mathcal{R}_2 \succeq \bot$	$Z_2 \succeq Y_0$
(3)	$\mathcal{R}_3 \succeq^* X_0$	$E_3 \succeq X_0^{\langle [\vert], 1\rangle}$ $Es_3 \succeq X_0^{\langle\rangle}$
(5)	$\mathcal{R}_5 \succeq \bot$	$Z_5^{\langle [\vert], 1\rangle} \succeq E_3$ $Z_5^{\langle\rangle} \succeq R_4$

Finally, handling a procedure $p(X_1, \ldots, X_n)$ call involves retrieving the constraints for the called procedure p from the denotation and projecting these onto the set of variables $\mathcal{A}rgs(p) \cup \{\mathcal{R}_{\eta_b}\}$. This projection operation makes sure that the constraints on these variables are in normal form, i.e. that they are expressed in terms of $\mathtt{in}(p)$. The resulting set of constraints is then renamed to the context of the call. The formal arguments of p, $\mathcal{A}rgs(p)$ are renamed to their corresponding actual argument in $\langle X_1, \ldots, X_n\rangle$. The constraints on $\mathcal{R}_{\eta_b}$ are renamed to constraint on $\mathcal{R}_\eta$, expressing that the call reduces to true, fail or code that is guaranteed to succeed if the body of the called procedure reduces to true, fail or code that is guaranteed to succeed.

Example 17. Let P denote the program consisting only of the definition of `append/3` depicted in Figure 3 and let (1) and (2) denote, respectively, the sets of constraints depicted in Examples 15 and 16. The fixed point computation for $\mathbf{P}[\![P]\!]$ starts with an empty denotation and hence, in the first round of the computation, the recursive call does not introduce any constraints; the result of $\mathbf{C}[\![\texttt{append/3}]\!]\{\}$ is a denotation that maps `append/3` to the constraint set $(1) \cup (2)$. It is only in the second round, when the constraints are projected and renamed, that the recursive call adds the constraints

$$R_4 \succeq Y_0 \qquad R_4^{\langle([],1)\rangle} \succeq Es_3^{\langle([],1)\rangle} \qquad \mathcal{R}_4 \succeq^* Es_0$$

One can verify that in a next round no new constraints are introduced by the recursive call, and hence $\mathbf{P}[\![P]\!]$ results in a denotation that associates `append/3` to the union of the constraints derived above with the sets (1) and (2).

3.4 From constraints to annotations

Once we have computed $\mathbf{P}[\![P]\!]$, it suffices to have a set of binding-times for the input variables of a procedure p in order to compute the binding-times of the remaining variables in the definition of p, as well as the annotations that are associated with a particular atom in the definition of p. Let us first introduce the semantic domain *Call*, that we use to represent a call in the domain of binding-times:

$$\mathit{Call} = \{p(\beta_1, \ldots, \beta_n) \mid p/n \in \Pi \text{ and } \forall i : \beta_i \in \mathcal{BT}^+\}$$

To ease notation, we assume that such a call contains a binding-time for each argument (input as well as output). However, since these calls are used to represent the binding-times of the *input* arguments of the call only, we asume the binding-times of the output arguments to be $\bot$. We will denote elements of *Call* by a single greek letter π if the particular procedure/argument combination is irrelevant. We can now define the annotation of a procedure with respect to a particular call as follows:

Definition 15 (procedure annotation). *Given a denotation $d \in Den$ for a program P and a call $p(\beta_1, \ldots, \beta_n) \in Call$, the* procedure annotation (of a procedure $p \in Proc(P)$) induced by a call $p(\beta_1, \ldots, \beta_n)$ *is defined as the least solution σ of $(d\,p)$ in which $\sigma(X_i) = \beta_i$ for every $X_i \in \texttt{in}(p)$.*

Being a solution of the set of binding-time constraints associated to a procedure p, a procedure annotation not only provides binding-times for all program variables in p, but also maps every binding-time variable of the form $\mathcal{C}_\eta$ to either $\bot$ or $\top$, denoting respectively that the goal at program point η in the procedure's body should be evaluated during specialisation, or be residualised. Being a *least* solution, a procedure annotation contains the least dynamic binding-times while still satisfying the congruence relation. As such, a procedure annotation of a procedure p with respect to a call π represents control information for a specialiser as to how to treat each subgoal of the body of p, when a call to p is approximated by π.

A polyvariant analysis for a program P and an initial call $p(\beta_1, \ldots, \beta_n)$ can then be performed by first computing the procedure annotation σ of p induced by $p(\beta_1, \ldots, \beta_n)$ and consecutively computing, for every call $q(X_1, \ldots, X_m)$ that occurs at some program point η in the definition of p, the procedure annotation of q induced by $q(\sigma(X_{1_\eta}), \ldots, \sigma(X_{m_\eta}))$. This process is repeated recursively until no more abstract calls are encountered for which no procedure annotation has been constructed yet. In other words, a polyvariant annotation process for a program P with initial call π boils down to computing the abstract callset of (P, π): The set of abstractions of all calls that can possibly be encountered during evaluation of P with respect to a call that is abstracted by π. Formally, we define also this annotation process by a number of semantic functions that define the meaning of a program P with respect to an initial call π as a set of calls in the domain of binding-times.

Definition 16 (annotation semantics). *The* first-order *annotation semantics has semantic domain $Den_c : \wp(Call)$ and semantic functions*

$$\mathbf{P_c} : Prog \mapsto Call \mapsto Den_c$$

$$\mathbf{C_c} : Proc \mapsto Call \mapsto Den_c \mapsto Den_c$$

$$\mathbf{G_c} : Goal \mapsto Call \mapsto Den_c$$

defined in Figure 7.

$$\mathbf{P_c}[\![P]\!]\pi = \text{lfp}(\bigcup_{p \in Proc(P)} \mathbf{C_c}[\![p]\!]\pi)$$

$$\mathbf{C_c}[\![p(X_1, \ldots, X_n) \leftarrow B]\!]\pi S = \bigcup_{p(\beta_1, \ldots, \beta_n) \in S \cup \{\pi\}} \mathbf{G_c}[\![B]\!]p(\beta_1, \ldots, \beta_n)$$

$$\begin{array}{ll}
\mathbf{G_c}[\![not(G)]\!]\pi & = \mathbf{G_c}[\![G]\!]\pi \\
\mathbf{G_c}[\![G_1, G_2]\!]\pi & = \mathbf{G_c}[\![G_1]\!]\pi \cup \mathbf{G_c}[\![G_2]\!]\pi \\
\mathbf{G_c}[\![G_1; G_2]\!]\pi & = \mathbf{G_c}[\![G_1]\!]\pi \cup \mathbf{G_c}[\![G_2]\!]\pi \\
\mathbf{G_c}[\![\mathit{if}\, G_1\ \mathit{then}\, G_2\ \mathit{else}\, G_3]\!]\pi & = \mathbf{G_c}[\![G_1]\!]\pi \cup \mathbf{G_c}[\![G_2]\!]\pi \cup \mathbf{G_c}[\![G_3]\!]\pi \\
\mathbf{G_c}[\![q(Y_1, \ldots, Y_n)]\!]\pi & = \{q(\sigma_\pi(Y_1), \ldots, \sigma_\pi(Y_n))\}
\end{array}$$

and $\mathbf{G_c}[\![A]\!]\pi = \emptyset$ for any other atomic goal A and where σ_π denotes the procedure annotation induced by $\pi \in Call$.

Fig. 7. The annotation semantics

The definition of the semantic functions $\mathbf{P_c}$, $\mathbf{C_c}$ and $\mathbf{G_c}$ is straightforward. The semantic domain $Den_c = \wp(Call)$ represents the set of all abstract callsets. The semantics of a program P with respect to an initial call π is defined as the least fixed point of repeatedly computing the semantics of each procedure (by $\mathbf{C_c}$) in P within the context of this initial call and a (possibly incomplete) denotation containing the result of analysis so far. The analysis function $\mathbf{C_c}$ constructs a partial denotation for a particular procedure as the union of the denotations obtained by analysing the procedure's body goal with respect to every call to the procedure encountered so far. The semantics of an individual goal G in the body of a procedure p is defined with respect to a call π to p. The definition of $\mathbf{G_c}$ is straightforward, as it only collects the abstract calls encountered in the annotation of p induced by π. Note that the analysis is guaranteed to create a finite number of procedure annotations since every procedure has a finite number of arguments, every such argument can only be approximated by a finite number of binding-times, and hence only a finite number of call patterns can be constructed for a particular procedure.

3.5 On the modularity of the approach

In summary, the binding-time analysis we have developed so far is to be performed in two phases. The first phase of the process performs the data flow analysis in a symbolic way. A procedure is analysed independent of a particular call pattern, and the analysis handles procedure calls by projecting and renaming the constraints that are associated to the called procedure. For a program that is divided into several modules, this means that the constraint generating phase of the analysis can be performed one module at a time, bottom-up in the module hierarchy if we consider hierarchies without circularities. Reconsider the module hierarchy from Fig. 1. The result of bottom-up analysis of this hierarchy is depicted in Fig. 8. First, the modules at the bottom level, M_4 and M_5 are analysed. Since these modules do not import any other modules, they can be treated as regular programs, and we can simply compute $\mathbf{P}[\![M_4]\!]$ and $\mathbf{P}[\![M_5]\!]$. The rounded boxes in the figure denote the result of computing $\mathbf{P}[\![M]\!]$ for a particular module M. The shaded part of the box represent this denotation, restricted to the procedures from the module's interface. Subsequently, the modules M_2 and M_3 can be analysed, since their analysis only requires the constraints from the interface procedures of M_4, respectively M_4 and M_5. Computation of $\mathbf{P}[\![M_2]\!]$ and $\mathbf{P}[\![M_3]\!]$ can proceed as before, with the exception that the fixed point computation should not be started from the empty denotation, but rather from $\mathbf{P}[\![M_4]\!]$ and $\mathbf{P}[\![M_4]\!] \cup \mathbf{P}[\![M_5]\!]$ respectively. Finally, once the results of analysing M_2, M_3 and M_5 are available, the module M_1 can be analysed. Note that in this process, each module is analysed only once. If a module, like M_5 in the example, is imported in more than one module, analysing the latter modules only requires the *result* of analysing the former.

The second phase of the analysis, computing the procedure annotations, is naturally a call-dependent process. Consequently, annotating a multi-module program for an initial call to a procedure p in the top-level module requires the

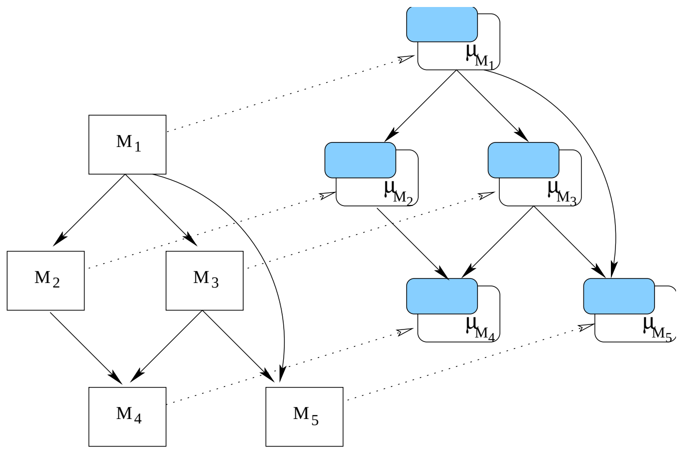

Fig. 8. Bottom-up analysis of the module hierarchy.

constraints for all the procedures (spread out over all modules) that are in the call graph for p. One could argue that this corresponds to analysing a multi-module program as if it was a single-module monolithic program. However, it should be noted that computing a procedure annotation induced by a particular call is a rather cheap process. Since the involved constraints are in normal form, it merely consists of performing a substitution on the right-hand side of the constraints and computing their least upper bounds. The hard part of the analysis – tracing the data flow between the input- and output arguments of a procedure – which possibly involves procedure calls over module boundaries, is done at the symbolic level, in a modular fashion.

4 Higher-order Binding-time Analysis

Mercury is a higher-order language in which *closures* can be created, passed as arguments of predicate calls, and in turn be called themselves. To describe the higher-order features of the language, it suffices to extend the definition of superhomogeneous form (see Definition 8) with two new kinds of atoms:

- A *higher-order unification* which is of the form $X \Leftarrow p(V_1, \ldots, V_k)$ where $X, V_1, \ldots, V_k \in \mathcal{V}$ and $p/n \in \Pi$ with $k \leq n$.
- A *higher-order call* which is of the form $X(V_{k+1}, \ldots, V_n)$ where X and $V_{k+1}, \ldots, V_n \in \mathcal{V}$ with $0 \leq k \leq n$.

A higher-order unification $X \Leftarrow p(V_1, \ldots, V_k)$ constructs a closure from an n-arity procedure p by *currying* the first k arguments (with $k \leq n$). The result of the construction is assigned to the variable X and denotes a procedure of arity $n - k$. Such a closure can be called by a higher-order call of the form $X(V_{k+1}, \ldots, V_n)$ where $V_{k+1}, \ldots, V_n$ are the $n - k$ remaining arguments. The effect of evaluating the conjunction $X \Leftarrow p(V_1, \ldots, V_k), X(V_{k+1}, \ldots, V_n)$ equals the effect of evaluating $p(V_1, \ldots, V_n)$.[6]

In order to represent higher-order *types* it suffices to add a special type constructor, *pred*, to $\Sigma_{\mathcal{T}}$. This constructor is special in the sense that it can be used with any arity and it has no type rule associated with it. Consequently, a higher-order type corresponds with a leaf node in a type tree. In what follows we represent higher-order types as $pred(t_1, \ldots, t_k)$ with $t_1, \ldots, t_k$ first-order types. We furthermore assume that higher-order types are not used in the definition of other types; that is, values of higher-order type are only constructed, called, or passed around as arguments of a procedure call.[7]

The basic problem when analysing a procedure involving higher-order calls, is that the control flow in the procedure is determined by the values of the higher-order variables. To retrieve a set of suitable binding-time constraints between the in- and output arguments of a higher-order call $X(Y_{k+1}, \ldots, Y_n)$, it is necessary to know to some extent to what closures X can be bound to during specialisation. Consequently, to achieve an acceptable level of precision, the symbolic data flow analysis needs to be enhanced by some form of *closure analysis* [23, 35] which basically computes for every higher-order call an approximation of the closures that may be bound to the higher-order variable involved. In what follows, we first define a suitable representation for such closure information; next we reformulate the first phase of our binding-time analysis in such a way that it integrates the derivation of closure information with the derivation of binding-time constraint systems. Doing so basically transforms the process of building constraint systems into a call dependent process, since closures can be passed around by procedure calls and hence the analysis needs to take the closure information from a particular call pattern into account. We conclude this section with a discussion on the modularity of the higher-order approach.

4.1 Representing closures

In order to use closures during binding-time analysis, where concrete values of the closure's curried arguments are approximated by binding-times, we introduce the notion of a *binding-time closure* as follows.

Definition 17 (binding-time closure). *A* binding-time closure *is a term of the form $p(\beta_1, \ldots, \beta_k)$ where $p/n \in \Pi$, $k \leq n$ and $\beta_1, \ldots, \beta_k \in \mathcal{BT}^+$. The set of all such binding-time closures is denoted by $\mathcal{C}los$.*

[6] When writing Mercury code, the programmer can also use lambda expressions to construct closures. These can, however, be converted into a regular procedure definition which is then again used to construct the closure as above. The Melbourne Mercury compiler does this conversion as part of the translation into superhomogeneous form. Note that closures cannot be constructed from other closures: once a closure is created, one can only call it or pass it as an argument to another procedure.

[7] In fact, this is also a limitation of release 0.9 of the Mercury implementation [40].

If $p/n \in \Pi$, $p(\beta_1, \ldots, \beta_k)$ approximates a set of procedures of arity $n-k$, each being an instance of p in which the first k arguments are fixed and whose values are approximated by the binding-times $\beta_1, \ldots, \beta_k$.

Example 18. Given the traditional `append/3` procedure and β_l being a binding-time approximating terms of type `list(T)` that are instantiated at least up to a list skeleton, *append*, $append(\beta_l)$ and $append(\bot, \beta_l)$ are examples of binding-time closures of arity 3, 2 and 1 respectively.

In order to obtain a precise binding-time analysis, we approximate the value of a higher-order variable with a *set* of binding-time closures. A singleton set $\{c\}$ describes that the higher-order variable under consideration is, during specialisation, definitely bound to a closure that is approximated by c. In general, a set $\{c_1, \ldots, c_n\}$ describes that the higher-order variable under consideration is bound during specialisation to a closure that is approximated either by c_1, $c_2, \ldots$, or c_n. To make this representation explicit, we alter the definition of the domain $\mathcal{B}$. Instead of containing only the values *static* and *dynamic*, we now include a value $static(S)$ with S being a set of binding-time closures. Note that, if we define $dynamic > static$ as before and $static(S_1) > static(S_2)$ if and only if $S_1 \supseteq S_2$, $\mathcal{B}$ is still partially ordered. Since the binding-times now include higher-order binding-times, we alter the definition of the partial order relation on $\mathcal{BT}$:

Definition 18 (covers). *Let $\beta, \beta' \in \mathcal{BT}$ such that $dom(\beta) \subseteq dom(\beta')$ or $dom(\beta') \subseteq dom(\beta)$. We say that β* covers *β', denoted by $\beta \succeq \beta'$ if and only if it holds for all $\delta \in dom(\beta) \cap dom(\beta')$ that:*

- *$\beta'(\delta) = dynamic$ implies $\beta(\delta) = dynamic$, and*
- *$\beta'(\delta) = static(S')$ implies $\beta(\delta) = static(S)$ and $S \supseteq S'$.*

Note that, with this new definition, the covers relation remains only defined between two binding-times that are derived from types that are instances of each other. In case of higher-order binding-times this means that both sets of binding-time closures contain closures of identical arity and argument types. Like before, we denote with $\mathcal{BT}^+$ the set $\mathcal{BT} \cup \{\top, \bot\}$, and $(\mathcal{BT}^+, \succeq)$ forms a complete lattice.

4.2 Higher-order binding-time analysis

We now reformulate the analysis from Section 3 such that it takes the higher-order constructs of Mercury into account. As a first observation, note that the binding-time constraints that are associated to first-order unifications and structured goals (see Figures 5 and 6) remain unchanged in the context of a higher-order analysis. To deal with higher-order constructions, we add an extra form of binding-time constraint to $\mathcal{BTC}$; namely a constraint of the form $X_\eta \succeq p(X_1, \ldots, X_k)$. The intended meaning is that the (higher-order) binding-time associated to X at program point η should at least contain a closure constructed from p and the binding-times of its arguments at program point η.

Formally, we extend the definition of a solution (Definition 10) such that for every constraint of the form $X_\eta \succeq p(X_{1_\eta}, \ldots, X_{k_\eta})$ it holds that

$$\sigma(X_\eta) \succeq \mathit{static}(\{p(\beta_1, \ldots, \beta_k)\}) \text{ where } \beta_i \succeq \sigma(X_{i_\eta}) \text{ for } 1 \leq i \leq k.$$

The main difference with the symbolic data flow analysis of Section 3 in a higher-order setting is that a set of constraints can no longer be associated to a procedure symbol (as in the semantic domain *Den*) because a typical higher order predicate is passed a procedure as one of its input arguments (e.g., a call to *map* has as one of its inputs the predicate to be applied on the elements of the list it has to process) and the resulting set of binding time constraints depends on the input predicate. Instead, in the higher-order analysis, we associate a set of binding-time constraints with a particular abstract call. Therefore, we define the analysis as an abstract semantics as before, but over the new semantic domain

$$\mathit{Den}_{cc} : \mathit{Call} \mapsto \wp(\mathcal{BTC}).$$

The notion of a procedure annotation of a procedure p induced by a call $p(\beta_1, \ldots, \beta_n)$ is straightforwardly adapted for use with a denotation in Den_{cc} rather than in *Den*. Moreover, given two such mappings $f, g \in \mathit{Den}_{cc}$, we define $f \cup g$ as a mapping in Den_{cc} with $\mathit{dom}(f \cup g) = \mathit{dom}(f) \cup \mathit{dom}(g)$ and

$$\forall x \in \mathit{dom}(f \cup g) : (f \cup g)(x) = \begin{cases} f(x) \cup g(x) & \text{if } x \in \mathit{dom}(f) \cap \mathit{dom}(g) \\ f(x) & \text{if } x \in \mathit{dom}(f) \text{ and } x \notin \mathit{dom}(g) \\ g(x) & \text{if } x \in \mathit{dom}(g) \text{ and } x \notin \mathit{dom}(f) \end{cases}$$

The resulting analysis is a call-dependent analysis that is basically a combination of the call-independent and call-dependent analyses of Section 3.

Definition 19 (higher-order semantics).
The higher-order semantics *has semantic domain*

$$\mathit{Den}_{cc} : \mathit{Call} \mapsto \wp(\mathcal{BTC})$$

and semantic functions

$$\mathbf{P_{cc}} : \mathit{Prog} \mapsto \mathit{Call} \mapsto \mathit{Den}_{cc}$$

$$\mathbf{C_{cc}} : \mathit{Proc} \mapsto \mathit{Call} \mapsto \mathit{Den}_{cc} \mapsto \mathit{Den}_{cc}$$

$$\mathbf{G_{cc}} : \mathit{Goal} \mapsto \mathit{Call} \mapsto \mathit{Den}_{cc} \mapsto \mathit{Den}_{cc}$$

$$\mathbf{A_{cc}} : \mathit{Atom} \mapsto \mathit{Call} \mapsto \mathit{Den}_{cc} \mapsto \mathit{Den}_{cc}$$

defined in Figure 9.

Again, the meaning of a program is defined as a fixed point computation over the meaning of the individual procedures in the program given a binding-time abstraction of the call with respect to which the program must be specialised. Each procedure is analysed (by $\mathbf{C_{cc}}$) within the context of this initial call and a

$$\mathbf{P_{cc}}[\![P]\!]\pi = \text{lfp}(\bigcup_{p \in Proc(P)} \mathbf{C_{cc}}[\![p]\!]\pi)$$

$$\mathbf{C_{cc}}[\![p(X_1,\ldots,X_n) \leftarrow B]\!]\pi d = \bigcup_{p(\beta_1,\ldots,\beta_n)\in dom(d)\cup\{\pi\}} \mathbf{G_{cc}}[\![B]\!]p(\beta_1,\ldots,\beta_n)d$$

$$\begin{array}{l}
\mathbf{G_{cc}}[\![(G'_{\eta'}, G''_{\eta''})_\eta]\!]\pi d = \\
\qquad \mathbf{G_{cc}}[\![G'_{\eta'}]\!]\pi d \cup \mathbf{G_{cc}}[\![G''_{\eta''}]\!]\pi d \cup \{(\pi, CC_{\text{conj}}(\eta,\eta',\eta''))\} \\
\mathbf{G_{cc}}[\![not_\eta(G_{\eta'})]\!]\pi d = \\
\qquad \mathbf{G_{cc}}[\![G_{\eta'}]\!]\pi d \cup \{(\pi, CC_{\text{not}}(\eta,\eta'))\} \\
\mathbf{G_{cc}}[\![if_\eta\ G'_{\eta'}\ then\, G''_{\eta''}\ else\, vG'''_{\eta'''}]\!]\pi d = \\
\qquad \mathbf{G_{cc}}[\![G'_{\eta'}]\!]\pi d \cup \mathbf{G_{cc}}[\![G''_{\eta''}]\!]\pi d \cup \mathbf{G_{cc}}[\![G'''_{\eta'''}]\!]\pi d \cup \{(\pi, CC_{\text{if}}(\eta,\eta',\eta'',\eta'''))\} \\
\mathbf{G_{cc}}[\![(G'_{\eta'}; G''_{\eta''})_\eta]\!]\pi d = \\
\qquad \mathbf{G_{cc}}[\![G'_{\eta'}]\!]\pi d \cup \mathbf{G_{cc}}[\![G''_{\eta''}]\!]\pi d \cup \{(\pi, CC_{\text{disj}}(\eta,\eta',\eta''))\} \\
\mathbf{G_{cc}}[\![A_\eta]\!]\pi d = \\
\qquad \mathbf{A_{cc}}[\![A_\eta]\!]\pi d \cup \{(\pi, S)\} \\
\qquad\qquad \text{where } S = \{X_\eta \succeq X_{\eta'} \mid X \in \mathtt{in}(A), \eta' \in \mathtt{reach}(X,\eta)\})\}
\end{array}$$

$$\begin{array}{l}
\mathbf{A_{cc}}[\![U]\!]\pi d = \{(\pi, \mathbf{A}[\![U]\!]d)\} \text{ for a first-order unification } U \\
\mathbf{A_{cc}}[\![X \Leftarrow p(X_1,\ldots,X_k)_\eta]\!]\pi d = \{(\pi, \{X_\eta \succeq p(X_1,\ldots,X_n), \mathcal{R}_\eta \succeq \bot\})\} \\
\mathbf{A_{cc}}[\![q(Y_1,\ldots,Y_n)_\eta]\!]\pi d = S_1 \cup S_2 \text{ where} \\
\qquad S_1 = \{(q(\beta_1,\ldots,\beta_n), \{\})\} \\
\qquad S_2 = \{(\pi, \rho(\text{proj}_{\mathcal{A}rgs(q),\mathcal{R}_{\eta_b}}(d\, q(\beta_1,\ldots,\beta_n))) \sqcup \{Y_{i_\eta} \succeq \mathcal{C}_\eta \mid Y_i \in \mathtt{out}(q)\})\} \\
\qquad\qquad \text{with } \beta_i = \sigma_\pi(Y_{i_\eta}) \\
\mathbf{A_{cc}}[\![X(Y_{k+1},\ldots,Y_n)_\eta]\!]\pi d = S_1 \cup S_2 \text{ where} \\
\qquad S_1 = \text{if } \sigma_\pi(X_\eta) = \top \\
\qquad\qquad \text{then } \{(q(\top,\ldots,\top),\ \{\}) \mid q/m \in Proc(P) \text{ and } m \geq n-k\} \\
\qquad\qquad \text{else } \{(q(\beta_1,\ldots,\beta_n),\ \{\}) \mid q(\beta_1,\ldots,\beta_k) \in S\} \\
\qquad\qquad\qquad \text{where } \sigma_\pi(X_\eta) = static(S) \text{ and } \beta_i = \sigma_\pi(Y_i) \text{ for } k+1 \leq i \leq n \\
\qquad S_2 = \{(\pi, \bigcup_{q(\beta_1,\ldots,\beta_n)\in dom(S_1)} \rho(\text{proj}_V(d(q\beta_1,\ldots,\beta_n)))\sqcup \\
\qquad\qquad\qquad \{Y_{i_\eta} \succeq \mathcal{C}_\eta \mid Y_i \in \mathtt{out}(q)\})\} \text{ where } V = \mathcal{A}rgs(q) \cup \{\mathcal{R}_{\eta_b}\}
\end{array}$$

Fig. 9. The higher-order semantics

denotation (in Den_{cc}) representing the (possibly incomplete) results of analysis so far. The definition of $\mathbf{G_{cc}}$, defining the abstract meaning of a goal, is basically identical to the definition of $\mathbf{G}$ from Section 3, apart from the facts that (1) it threads a denotation as well as the abstract call to the procedure that is currently being analysed and (2) it associates this abstract call to the constraints for a particular goal. The same observations hold for the definition of $\mathbf{A_{cc}}$. The constraints derived for a first-order unification are identical to those derived by $\mathbf{A}$. A higher-order construction results in a constraint stating that the binding-time of the higher-order variable must contain at least the abstract closure created at this program point. Note that we propagate the binding even when the construction is under dynamic control, as this binding allows to substantially simplify the analysis of higher-order calls. Being a construction, reduction can never result in code that might fail during exeuction, hence the (superfluous) constraint on $\mathcal{R}_\eta$.

Handling procedure calls is somewhat more involved than in the first-order case. Retrieving the constraints associated to a first-order call from the denotation now requires to compute the binding-times of the arguments in of the call. As before, σ_π represents the procedure annotation induced by the call π. The binding-time variables in the resulting (projected) constraints are again renamed to the actual arguments of the call $X_1, \ldots, X_n$ and the control variable $\mathcal{R}_{\eta_b}$ is renamed to $\mathcal{R}_\eta$, as before. As for the other goals, the resulting constraints are associated to the abstract call π for which the surrounding procedure is being analysed. The resulting mapping, in Figure 9 denoted by S_2, is updated with the mapping $\{(q(\beta_1, \ldots, \beta_n), \{\})\}$ in order to make sure that the call $q(\beta_1, \ldots, \beta_n)$ is in the domain of the newly constructed denotation, and hence will be analysed during a next round of the analysis. Note that the use of $\cup$ guarantees that if the call was already in the domain of the donation, the set of constraints associated to it remains unchanged. A higher-order call is basically handled as a set of first-order calls. First, the binding-time of the higher-order variable is retrieved from the procedure annotation σ_π for the currently analysed procedure/call combination. If this binding-time equals $\mathit{static}(S)$, each closure $q(\beta_1, \ldots, \beta_k) \in S$ is transformed to a first-order call by adding $\sigma_\pi(X_{k+1}), \ldots, \sigma_\pi(X_n)$ to its arguments. From then on, the call is handled as a first-order call. The constraints associated to this call are retrieved from the denotation and added to the denotation under construction, and the call itself is added to the domain of the denotation under construction.

4.3 On the modularity of the approach

In a higher-order setting, the constraint generation phase of our binding-time analysis is a call dependent process. Indeed, the data flow dependencies in a procedure are determined by the closures contained in the procedure's call pattern. This suggests that the advantage of modularity, associated to the constraint based technique in a first-order setting, might no longer hold in a higher-order setting. However, to some extent the analysis can still be performed in a bottom-up, modular way. For a module M that exports the predicates $p_1, \ldots, p_n$ we initiate the analysis with:

$$\bigcup_{p \in \{p_1, \ldots, p_n\}} \mathbf{P_{cc}}[\![P]\!]p(\top, \ldots, \top).$$

At first sight, it might seem strange to perform a call-dependent analysis with respect to an inital call in which all arguments are approximated by $\top$. However, recall that only the higher-order parts of the call patterns influence the resulting constraint systems. Hence, for those procedures that have no higher-order arguments, the constraint system derived by the call dependent analysis for a call $p(\top, \ldots, \top)$ equals the one derived by the call independent analysis of Section 3, and it can readily be used by other modules importing these procedures. Note that the call dependent nature of the process ensures that closure information that is constructed in a module M, is propagated inside M itself. It is only if closure information is "lost" over a module boundary that the resulting analysis is less precise than a full call dependent analysis over the complete multi-module program. This is the case when, in some module, closure information is available in some arguments of a call to an imported procedure p whereas, being imported, the constraints that are used for p are those obtained by analysing $p(\top, \ldots, \top)$.

5 Example

In this section, we present an example, and use it to discuss to what extent the proposed analysis is also applicable in the context of Prolog.

5.1 A simple interpreter

Consider the simple interpreter for arithmetic expressions depicted in Figure 10, adapted from a Prolog version discussed and specialized in a companion chapter [29]. The program consists of a number of type definitions and two predicates. The type `env` defines an environment as a list of elements, each element being a pair (type `elem`) consisting of an identifier (type `ident`) and an integer (type `int`). We assume that the types `ident` and `int` are atomic and builtin. The type `exp` defines an expression as either a constant integer, a variable denoted by an identifier, or the sum of two expressions.

The predicate `lookup/3` takes an identifier and an environment as input, searches the value associated to the identifier in the environment and returns this value or fails. Note that the predicate is defined as being non-deterministic in order to mimick a purely declarative implementation in Prolog. The interpreter itself is represented by the predicate `int` which takes an expression and an environment as input and returns the value of the expression or fails. Both predicates are given in superhomogeneous form.

```
:- type env --> nil ; cons(elem, env).
:- type elem --> pair(ident,int).

:- type exp --> cst(int) ; var(ident) ; +(exp,exp).

:- pred lookup(ident, env, int).
:- mode lookup(in,in,out) is multi.

lookup(V,E,Val):- E$\Rightarrow_1$ cons(A,As), A$\Rightarrow_2$ pair(I,VI), (
     V$==_3$ I, Val$:=_4$ VI
     ;
     lookup(V,As,T)$_5$, Val$:=_6$ T).

:- pred int(exp,env,int).
:- mode int(in,in,out) is multi.

int(E,Env,R):-(
     E$\Rightarrow_1$ cst(C), R$:=_2$ C
     ;
     E$\Rightarrow_3$ var(V), lookup(V,Env,Val)$_4$, R$:=_5$ Val
     ;
     E$\Rightarrow_6$ +(A,B), int(A,Env,R1)$_7$, int(B,Env,R2)$_8$, plus(R1,R2,R)$_9$).
```

Fig. 10. A simple interpreter

After call-independent analysis, the binding-time constraints associated with the `lookup/3` predicate are as follows.

$$\begin{array}{l}
A \succeq E^{\langle(cons,1)\rangle} \\
As \succeq E \\
I \succeq E^{\langle(cons,1),(pair,1)\rangle} \\
I \succeq^* E \\
VI \succeq E^{\langle(cons,1),(pair,2)\rangle} \\
VI \succeq^* E \\
Val_4 \succeq E^{\langle(cons,1),(pair,2)\rangle} \\
Val_4 \succeq^* E \sqcup E^{\langle(cons,1)\rangle} \sqcup E^{\langle(cons,1),(pair,1)\rangle} \sqcup V \\
T \succeq E^{\langle(cons,1),(pair,2)\rangle} \\
T \succeq^* E \sqcup E^{\langle(cons,1)\rangle} \sqcup E^{\langle(cons,1),(pair,1)\rangle} \sqcup V \\
Val_6 \succeq E^{\langle(cons,1),(pair,2)\rangle} \\
Val_6 \succeq^* E \sqcup E^{\langle(cons,1)\rangle} \sqcup E^{\langle(cons,1),(pair,1)\rangle} \sqcup V
\end{array}$$

All constraints are in normalised form. Where relevant, a binding-time variable is indexed by a subscript indicating the program point at which the constraint holds. Recall that the $\succeq$-constraints express the regular data flow, whereas the $\succeq^*$-constraints reflect the specialisation-strategy: a constraint $X \succeq^* Y^\delta$ denotes that the binding-time of X cannot be static if the node δ in the binding-time of Y is marked *dynamic*. Such a constraint is due to the presence, earlier in the predicate, of a deconstruction (or test) on Y^δ that may be residualised and subsequently fail at run-time. The interpretation of these constraints is as follows. The data-flow (or $\succeq$) constraints are obtained in a straightforward way, by projecting the constraints obtained from the unifications. The strategy (or $\succeq^*$) constraints are somewhat more involved. The constraints $I \succeq^* E$ and $VI \succeq^* E$ denote that I and VI must be $\top$ in case E is not bound to an outermost functor. Indeed, if E is not bound to an outermost functor, the deconstruction at program point 1 cannot be reduced at specialisation-time and the atom at program point 2 (in which I and VI are assigned their value) is under dynamic control and hence cannot be reduced at specialisation time. Subsequently, the construction at program point 4 is under dynamic control if one of the preceeding atoms cannot be reduced or results in code that may fail at runtime, which is the case if either the environment E, the elements of the environment ($E^{\langle(cons,1)\rangle}$), the identifiers within each such element ($E^{\langle(cons,1),(pair,1)\rangle}$ or the variable V is not bound to an outermost function. Similar considerations explain the $\succeq^*$ constraints on T and Val at program point 6 in the other branch of the disjunction. The constraints on T are equal to the least upper bound of those (in the least fixed point) on Val_4 and Val_6. Recall that the constraints on T, which originate from the recursive call, are obtained from $T \succeq Val_4 \sqcup Val_6$.

The binding-time constraints derived for the `int/3` predicate are as follows.

$$
\begin{array}{l}
C \succeq E^{\langle(cst,1)\rangle}\\
R_2 \succeq E^{\langle(cst,1)\rangle}\\
R_2 \succeq^* E\\
V \succeq E^{\langle(var,1)\rangle}\\
Val \succeq Env^{\langle(cons,1),(pair,2)\rangle}\\
Val \succeq^* Env \sqcup Env^{\langle(cons,1)\rangle} \sqcup Env^{\langle(cons,1),(pair,1)\rangle} \sqcup E^{\langle(var,1)\rangle}\\
R_5 \succeq Env^{\langle(cons,1),(pair,2)\rangle}\\
R_5 \succeq^* E \sqcup Env \sqcup Env^{\langle(cons,1)\rangle} \sqcup Env^{\langle(cons,1),(pair,1)\rangle} \sqcup E^{\langle(var,1)\rangle}\\
A \succeq E^{\langle(+,1)\rangle}\\
B \succeq E^{\langle(+,2)\rangle}\\
R1 \succeq E^{\langle(+,1),(cst,1)\rangle} \sqcup Env^{\langle(cons,1),(pair,2)\rangle}\\
R1 \succeq^* E \sqcup E^{\langle(+,1)\rangle} \sqcup E^{\langle(+,2)\rangle} \sqcup E^{\langle(+,1),(var,1)\rangle} \sqcup E^{\langle(+,2),(var,1)\rangle} \sqcup\\
\qquad Env \sqcup Env^{\langle(cons,1)\rangle} \sqcup Env^{\langle(cons,1),(pair,1)\rangle}\\
R2 \succeq E^{\langle(+,2),(cst,1)\rangle} \sqcup Env^{\langle(cons,1),(pair,2)\rangle}\\
R2 \succeq^* E \sqcup E^{\langle(+,1)\rangle} \sqcup E^{\langle(+,2)\rangle} \sqcup E^{\langle(+,1),(var,1)\rangle} \sqcup E^{\langle(+,2),(var,1)\rangle} \sqcup\\
\qquad Env \sqcup Env^{\langle(cons,1)\rangle} \sqcup Env^{\langle(cons,1),(pair,1)\rangle}\\
R_9 \succeq E^{\langle(+,1),(cst,1)\rangle} \sqcup E^{\langle(+,2),(cst,1)\rangle} \sqcup Env^{\langle(cons,1),(pair,2)\rangle}\\
R_9 \succeq^* E \sqcup E^{\langle(+,1)\rangle} \sqcup E^{\langle(+,2)\rangle} \sqcup E^{\langle(+,1),(var,1)\rangle} \sqcup E^{\langle(+,2),(var,1)\rangle} \sqcup\\
\qquad Env \sqcup Env^{\langle(cons,1)\rangle} \sqcup Env^{\langle(cons,1),(pair,1)\rangle}
\end{array}
$$

These constraints are obtained in a similar way as those for the `lookup` predicate.

Assume we want to specialise this program for the query

```
int(+(cst(2),+(var(x),cst(3))), [pair(y,Yval),(x,Xval)],Res)  (1)
```

i.e., the expression to compute is fully instantiated and the domain of the environment mapping is fully defined but the concrete values associated to the identifiers are as yet unknown. These degrees of instantiation are expressed by the binding-times β_{exp} defined for the type `exp` and β_{env} defined for the type `env`.

$$\beta_{exp} = \left\{ \begin{array}{lll} (\langle\rangle, \mathit{static}), & (\langle(cst, 1)\rangle, \mathit{static}), & (\langle(var, 1)\rangle, \mathit{static}) \\ (\langle(+, 1)\rangle, \mathit{static}), & (\langle(+, 2)\rangle, \mathit{static}) & \end{array} \right\}$$

$$\beta_{env} = \left\{ \begin{array}{l} (\langle\rangle, \mathit{static}) \\ (\langle(cons, 1), (pair, 1)\rangle, \mathit{static}) \\ (\langle(cons, 1), (pair, 2)\rangle, \mathit{dynamic}) \end{array} \right\}$$

Note that the abstract call `int(`β_{exp}`,`β_{env}`,_)` will give rise to an abstract call `lookup(`*static*`,`β_{env}`,_)`. In the least solution of the constraints for `lookup` with respect to this call, we obtain that the output argument $Val = Val_4 \sqcup Val_6 =$ *dynamic*. However, the input to each test or deconstruction in `lookup` is at least bound to an outermost functor and hence is a candidate for reduction. In addition, if we look at the strategy constraints

$$\begin{array}{l} \mathcal{C}_2 \succeq^* E \\ \mathcal{C}_3 \succeq^* E \sqcup E^{\langle(cons,1)\rangle} \\ \mathcal{C}_4 \succeq^* E \sqcup E^{\langle(cons,1)\rangle} \sqcup V \sqcup E^{\langle(cons,1),(pair,1)\rangle} \\ \mathcal{C}_5 \succeq^* E \sqcup E^{\langle(cons,1)\rangle} \\ \mathcal{C}_6 \succeq^* E \sqcup E^{\langle(cons,1)\rangle} \sqcup V \sqcup E^{\langle(cons,1),(pair,1)\rangle} \end{array}$$

we derive that none of the atoms is under dynamic control and consequently, each atom can be annotated as reducible.

Consequently, for the `int` predicate we obtain $R =$ *dynamic* but similarily to the case of the `lookup` predicate, none of the atoms is under dynamic control and the input to each unification is bound to at least an outermost constructor. Hence all unifications can be reduced. Only the predicate `plus`, which we assume builtin, has both input arguments *dynamic* and need to be residualised. The result of specialisation using the obtained annotations is the residual program

```
int(Xval,Yval,Res):- plus(Xval,3,T), plus(2,T,Res).
```

5.2 The Prolog case

The basic characteristic of Mercury that make this work feasible is the presence of type- and mode information. Hence, one may ask to what extent the technique can be carried over to the analysis of (pure) Prolog programs. Let us assume that the same type information as above is available. Given that the normal use of the `int/3` predicate is with mode `(i,i,o)`, a mode analysis is able to show that `lookup/3` is also called with mode `(i,i,o)` and that both predicates return a ground answer. Taking care that variables in output positions of predicates are first occurrences (hence free variables) one can obtain a normalisation that is almost a replica of the Mercury code.

```
lookup(V,E,Val):-E=cons(A,As), A=pair(I,VI), V=I, Val=VI.
lookup(V,E,Val):-E=cons(A,As), A=pair(I,VI), lookup(V,As,T),
                 Val=T.


int(E,Env,R):-E = cts(C), R=C.
int(E,Env,R):-E = var(V), lookup(E,Env,Val), R=Val .
int(E,Env,R):-E = +(A,B), int(A,Env,R1), int(B,Env,R2),
              is+(R1,R2,R).
```

Using the mode information about the variables participating in unifications, one could classify them into tests, assignments, constructions and deconstructions as in the Mercury code. There is one difference. In the case of Mercury, assignments and constructions are guaranteed to succeed. In the case of our mode analysis, a variable not having mode input can still be partially intantiated, hence the unfication could fail at run-time. This will not happen in the example at hand. Indeed a simple local analysis shows that the variables being assigned are effectively free. E.g. in `Val=VI`, Val is the first occurrence of the output variable. Whether a unification η can fail has to be properly encoded in the special binding-time analysis variable $\mathcal{R}_\eta$. Apart from this, given the type information and the specification of the query to be specialised, the binding time analysis as done for Mercury can be performed, leading to the same annotations and hence, a specialiser as LOGEN [30] could derive the same specialised code.

Finally, it is feasible to handle more complex modes than simply input and output. In [7], a more refined mode analysis, called rigidity analysis is developed. Given a term t of type τ, it considers all subtypes τ' of τ. The term is τ'-rigid if it cannot have a well-typed instance that has a variable as a subterm of type τ'. Such a type based rigidity analysis can provide more detailed mode information that has the potential to contribute to a better binding-time analysis. For example, such an analysis could show that a term of type `elem` (cnfr. the simple interpreter) that is not ground, is `ident`-rigid.

To conclude the discussion of this example, we note that — within the context of Prolog – the results obtained by the binding-time analysis could be directly fed to the LOGEN offline partial deduction system [25, 30]. This system uses the notion of a *binding-type* to characterise specialisation-time values. Basic binding-types are *static* — characterising a value as ground — and *dynamic* – characterising a value as possibly non-ground – but more involved binding-types can be declared by the user using binding-type rules, much in the same way as types are declared by type rules.

In the interpreter example, the binding-times β_{exp} and β_{env} could be translated to the following binding-type definitions:

```
:- type exp ---> cst(static) ; var(static) ; +(static,static).

:- type elem --> pair(static,dynamic).
:- type env ---> nil ; cons(elem,env).
```

Input to the LOGEN system would then consist of the program in which every call is annotated as reducible (by means of the `unfold` annotation [25, 30]) together with the binding-type classification of the query `int(exp,env,dynamic)`. In the companion chapter [29] we present in more detail how this example program can be specialized using the LOGEN system and the so-derived annotations. Further work is needed to investigate whether our binding-time analysis can be adapted for the Prolog setting with LOGEN's binding-types.

6 Discussion

Constraint based (binding-time) analysis has been considered before. In [17], Henglein develops such a constraint-based (higher-order) binding-time analysis for λ-calculus by viewing the problem as a type inference problem for annotated λ-terms in a two-level λ-calculus. A set of constraints capturing local binding-time requirements is created and transformed into a normal form. A solver is used to find a consistent minimal binding-time classification. The analysis is re-developed, concentrating on the aspect of polyvariance, for a PCF-like language in [19]. Henglein's analysis is scaled up by Bondorf and Jørgensen in [5], where they construct three (monovariant) analyses to be used in the partial evaluator Similix [4]. An important conceptual advantage, mentioned among others in [5], of doing binding-time analysis by constraint normalisation is the fact that the constraint based approach is viewed as a more *elegant* description of the analysis, compared with a direct abstract interpretation approach in which the source code is abstractly interpreted over the domain of binding-times. Indeed, in the constraint-based approach, problem and solution are separated: the constraint system expresses the binding-time *requirements* on the involved variables, whereas actual binding-times are contained in a *solution* to the constraint system. A practical consequence of this separation is that the data flow analysis, being performed at the symbolic level, needs to be performed only once for each predicate (in a first-order setting) rather than performing a separate analysis for every (abstract) call to the predicate. This result extends – at least to some extent – to a higher-order setting in the sense that the data flow analysis needs to be performed only once for each combination of a predicate with the closure information from its arguments.

In this work, we have shown that a constraint-based approach is also feasible for the logic programming language Mercury. The available type information allows to construct a precise domain of binding-times, whereas the available mode information allows to express the data flow constraints in a sufficiently precise way. Apart from being modular, the resulting analysis is polyvariant, and able to deal with partially instantiated data structures. A prototype implementation of the analysis was made and in [49] we describe some experiments that show the practical feasibility of the analysis. An interesting topic for further research is to couple the binding-time analysis with an offline specialiser and to perform experiments to determine the obtainable speedups.

Strongly related to our domain of binding-times is the domain proposed and used by Launchbury [28] who defines a system of types and derives a finite domain of *projections* over each type. Such a projection maps a value to a part of the value that is definitely static, as such "blanking" out the dynamic part. In recent work [3, 2], a binding-time analysis is presented for the lambda calculus that allows an expression to be both static and dynamic at the same time; the general idea is to be able to access statically the (static) components of a residualised data structure. The exact relation and/or integration with a fine-grained domain of binding-times as employed by our technique is an interesting topic for further research.

Upgrading binding-time analysis to deal with Mercury's higher-order constructs requires closure information. In the literature, also closure analysis has been formulated by means of abstract interpretation [4, 9] as well as by constraint solving [16, 35, 18]. Bondorf and Jørgensen [5] develop a constraint-based flow analysis that traces higher-order flow as well as flow of constructed (first-order) values. In this work, we have combined closure analysis with binding-time analysis and used constraints to express the first-order as well as the higher-order data flow. We have enhanced the domain of binding-times to include a set of closures that represents the binding-time of a higher-order value, and formulated the constraint-generation phase as a call dependent process in which however only the higher-order parts of the call pattern determine the result of the analysis. During constraint generation, the constraints involving higher-order values are evaluated, and the resulting closure information is used to decide what constraints to incorporate, possibly propagating closure information down into the called procedures.

We have discussed in detail how the analysis can be applied to multi-module programs according to a one module at a time scenario in Sections 3.5 and 4.3. If we do not wish to propagate closure information over module boundaries, the constraint generation phase can be performed one module at a time, bottom-up in the module hierarchy. Remaining issues are precisely such inter-module closure propagation and the handling of circularities in the module hierarchy. Recent work [8] presents a framework for the (call-dependent) analysis of multi-module programs that solves both problems. The key invariant in the approach of [8] is that at each stage of the process, the analysis results are correct, but reanalysis may – when more information is available – produce more accurate results. The analysis performs some extra bookkeeping such that, when a module is analysed, it records both the call patterns occurring in the calls to the imported procedures, and the analysis results of the module's exported procedures. When the recorded information contains new calls (or calls with a more accurate call pattern) to the imported modules, the analysis may decide to reanalyse the relevant imported modules with respect to the more accurate call patterns. Likewise, the recording of more accurate analysis results for a module's exported procedures can trigger the reanalysis of those modules that would possibly profit from these more accurate results. Note that our binding-time analysis neatly fits such an approach: initially, a module's exported procedures are analysed with respect to $\top$ (no closure information is available). The resulting binding-time constraint systems are correct, but could possibly be rendered more precise, when the procedures are (re)analysed with respect to a more accurate call pattern (one that *does* contain some closure information). To the best of our knowledge, the binding-time analysis of modular programs has been considered only occasionally before. Henglein and Mossin [19] note that a symbolic representation of binding-times allows a modular approach. Based on such a symbolic analysis, [12] present a method to specialise a multi-module program – written in a simple yet higher-order functional language – by constructing, for each of the modules, a generating extension, while using only the result of a call-independent binding-time analysis. The analysis assumes that annotations indicating whether a function must be unfolded are given by hand and is restricted to module hierarchies without circular dependencies.

To summarise, we can state that few binding-time analyses have been developed that are polyvariant, deal with partially instantiated data, modules *and* higher-order constructs for a realistic language. Our binding-time analysis achieves this for the Mercury language by combining a number of known techniques: partially instantiated structures are dealt with by incorporating a structured and precise domain of binding-times, polyvariance and modularity are achieved by computing the binding-times symbolically and higher-order information is incorporated by propagating closure information during the symbolic phase of the analysis. Two important limitations of our technique are in the modularity of the approach, in particular the lack of propagation of closure information over module boundaries and the handling of circularities in the module dependency graph. Fortunately, both issues can be addressed by imposing a system like [8] on top of our technique.

Acknowledgments

We would like to thank the anonymous referees for providing valuable feedback. We are also grateful to Kim Henriksen for his comments.

References

1. L.O. Andersen. Binding-time analysis and the taming of C pointers. In *PEPM93*, pages 47–58. ACM, 1993.
2. K. Asai. Binding-time analysis for both static and dynamic expressions. *New Generation Computing*, 20(1):27–52, 2001.
3. Kenichi Asai. Binding-time analysis for both static and dynamic expressions. In *Static Analysis Symposium*, pages 117–133, 1999.
4. A. Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, 17:3–34, 1991.
5. A. Bondorf and J. Jørgensen. Efficient analyses for realistic off-line partial evaluation. *Journal of Functional Programming*, 3(3):315–346, 1993.

6. 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.
7. Maurice Bruynooghe, Wim Vanhoof, and Michael Codish. Pos(T) : Analyzing dependencies in typed logic programs. In *Perspectives of System Informatics, 4th International Andrei Ershov Memorial Conference, PSI 2001, Revised Papers*, volume 2244 of *LNCS*, pages 406–420. Springer-Verlag, 2001.
8. 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 K.K. Lau, editor, *Preproceedings of LOPSTR 2000*, pages 64–71, 2000.
9. C. Consel. Binding time analysis for higher order untyped functional languages. In *1990 ACM Conference on Lisp and Functional Programming, Nice, France*, pages 264–272. ACM, 1990.
10. C. Consel et al. A uniform approach for compile-time and run-time specialization. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, pages 54–72. Springer-Verlag, 1996.
11. Thomas Conway, Fergus Henderson, and Zoltan Somogyi. Code generation for Mercury. In John Lloyd, editor, *Proceedings of the International Symposium on Logic Programming*, pages 242–256, Cambridge, 1995. MIT Press.
12. D. Dussart, R. Heldal, and J. Hughes. Module-sensitive program specialisation. In *SIGPLAN '97 Conference on Programming Language Design and Implementation, June 1997, Las Vegas*, pages 206–214. ACM, 1997.
13. Y. Futamura. Partial evaluation of a computation process — an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.
14. R. Glück and J. Jørgensen. An automatic program generator for multi-level specialization. *Lisp and Symbolic Computation*, 10:113–158, 1997.
15. C.K. Gomard and N.D. Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1(1):21–69, 1991.
16. N. Heintze. Set-based analysis of ML programs. In *ACM Conference on Lisp and Functional Programming*, pages 306–317, 1994.
17. F. Henglein. Efficient type inference for higher-order binding-time analysis. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991 (Lecture Notes in Computer Science, vol. 523)*, pages 448–472. ACM, Springer-Verlag, 1991.
18. F Henglein. Simple closure analysis. Technical Report D-193, DIKU Semantics Report, 1992.
19. F. Henglein and C. Mossin. Polymorphic binding-time analysis. In D. Sannella, editor, *Programming Languages and Systems — ESOP'94. 5th European Symposium on Programming, Edinburgh, U.K., April 1994 (Lecture Notes in Computer Science, vol. 788)*, pages 287–301. Springer-Verlag, 1994.
20. L. Hornof and J. Noyé. Accurate binding-time analysis for imperative languages: Flow, context, and return sensitivity. In *PEPM97*, pages 63–73. ACM, 1997.
21. Dean Jacobs and Anno Langen. Static analysis of logic programs for independent AND-parallelism. *Journal of Logic Programming*, 13(2 &3):291–314, May/July 1992.
22. G. Janssens and M. Bruynooghe. Deriving descriptions of possible values of program variables by means of abstract interpretation. *Journal of Logic Programming*, 13(2&3):205–258, 1992.
23. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

24. N.D. Jones, P. Sestoft, and H. Søndergaard. An experiment in partial evaluation: The generation of a compiler generator. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications, Dijon, France. (Lecture Notes in Computer Science, vol. 202)*, pages 124–140. Springer-Verlag, 1985.
25. 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.
26. Vitaly Lagoon, Fred Mesnard, and Peter Stuckey. Termination analysis with types is more accurate. In *Logic Programming, 19th International Conference, ICLP 2003,*, volume 2916 of *LNCS*, pages 254–268. Springer-Verlag, 2003.
27. T. K. Lakshman and Uday S. Reddy. Typed Prolog: A semantic reconstruction of the Mycroft-O'Keefe type system. In Kazunori Saraswat, Vijay; Ueda, editor, *Proceedings of the 1991 International Symposium on Logic Programming (ISLP'91)*, pages 202–220, San Diego, CA, 1991. MIT Press.
28. J. Launchbury. Dependent sums express separation of binding times. In K. Davis and J. Hughes, editors, *Functional Programming, Glasgow, Scotland, 1989*, pages 238–253. Springer-Verlag, 1990.
29. Michael Leuschel, Stephen J. Craig, Maurice Bruynooghe, and Wim Vanhoof. Specializing interpreters using offline partial deduction. In K.K. Lau and M. Bruynooghe, editors, *Program Development in Computational Logic*. Springer-Verlag, 2004.
30. Michael Leuschel, J. Jørgensen, Wim Vanhoof, and Maurice Bruynooghe. Offline specialisation in Prolog using a hand-written compiler generator. *Theory and Practice of Logic Programming*, 4(1):139–191, 2002.
31. J. W. Lloyd. Declarative error diagnosis. *New Generation Computing*, 5:133–154, 1987.
32. 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, 1989.
33. 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.
34. A. Mycroft and R. A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23(3):295–307, 1984.
35. Jens Palsberg. Closure analysis in constraint form. *ACM Transactions on Programming Languages and Systems*, 17(1):47–62, 1995.
36. G. Puebla and M. Hermenegildo. Some issues in analysis and specialization of modular Ciao-Prolog programs. In M. Leuschel, editor, *Proceedings of the Workshop on Optimization and Implementation of Declarative Languages*, Las Cruces, 1999. In Electronic Notes in Theoretical Computer Science, Volume 30 Issue No.2, Elsevier Science.
37. Jan-Georg Smaus. Analysis of polymorphically typed logic programs using ACI-unification. In Robert Nieuwenhuis and Andrei Voronkov, editors, *8th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 2250 of *Lecture Notes in Artificial Intelligence*, pages 280–295. Springer-Verlag, 2001.
38. Jan-Georg Smaus, Patricia M. Hill, and Andy King. Mode Analysis Domains for Typed Logic Programs. In *Logic Program Synthesis and Transformation*, pages 82–101, 1999.

39. Z. Somogyi. A system of precise modes for logic programs. In *International Conference on Logic Programming*, pages 769–787, 1987.
40. Zoltan Somogyi et al. The Melbourne Mercury compiler, release 0.9.
41. Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The implementation of Mercury, an efficient purely declarative logic programming language. In *Proceedings of the ILPS'94 Postconference Workshop on Implementation Techniques for Logic Programming Languages*, 1994.
42. Zoltan Somogyi, Fergus Henderson, and Thomas Conway. Logic programming for the real world. In *Proceedings of the ILPS'95 Postconference Workshop on Visions for the Future of Logic Programming*, 1995.
43. 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, 1996.
44. Zoltan Somogyi, Fergus Henderson, Thomas Conway, Andres Bromage, Tyson Dowd, David Jeffery, Peter Ross, Peter Schachte, and Simon Taylor. Status of the Mercury system. In *Proceedings of the JICSLP'96 Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages*, 1996.
45. P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Type analysis of Prolog using type graphs. *Journal of Logic Programming*, 22(3):179–209, 1995.
46. W. Vanhoof. Binding-time analysis by constraint solving: a modular and higher-order approach for Mercury. In M. Parigot and A. Voronkov, editors, *Logic for Programming and Automated Reasoning, 7th Intl. Conference, LPAR2000*, volume 1955 of *Lecture Notes in Artificial Intelligence*, pages 399 – 416, Reunion Island, France, 2000.
47. 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.
48. W. Vanhoof and M. Bruynooghe. Binding-time analysis for Mercury. In *Pre-proceedings of LOPSTR'99*, Venice, Italy, 1999. Technical Report, University of Venice.
49. Wim Vanhoof. *Techniques for on- and off-line specialisation of logic programs.* Phd, Department of Computer Science, K.U.Leuven, Leuven, Belgium, jun 2001. Pages: xiv+323+xxxiii.
50. Wim Vanhoof and Maurice Bruynooghe. Binding-time annotations without binding-time analysis. In *Logic for Programming, Artificial Intelligence, and Reasoning, 8th International Conference, Proceedings*, volume 2250 of *Lecture Notes in Artificial Intelligence*, pages 707–722. Springer-Verlag, 2001.
51. Wim Vanhoof and Maurice Bruynooghe. When size does matter - Termination analysis for typed logic programs. In *Logic-based Program Synthesis and Transformation, 11th International Workshop, LOPSTR 2001, Selected Papers*, volume 2372 of *Lecture Notes in Computer Science*, pages 129–147. Springer-Verlag, 2002.