Towards memory reuse in Mercury

Nancy Mazur, Gerda Janssens, and Maurice Bruynooghe

Department of Computer Science, K.U.Leuven
Celestijnenlaan, 200A, B–3001 Heverlee, Belgium
nancy,gerda,maurice@cs.kuleuven.ac.be

Abstract. While Mercury allows destructive input/unique output modes which direct the compiler to reuse memory, use of these modes is very cumbersome for the programmer. Moreover it does not fit the declarative programming paradigm where the programmer doesn't have to worry about the details of memory management.

The paper briefly reports on some experiments with a prototype analyser which aims at detecting memory available for reuse. The prototype is based on the live-structure analysis developed by us for logic programs extended with declarations.

Yet the major contribution of this paper consists of the development of the principles of a module based analysis which are essential for the analysis of large Mercury programs with code distributed over many modules.

1 Introduction

Logic programs do not have destructive assignment. It is one of the cornerstones of their declarativeness. However, the absence of destructive assignment has an implementation cost; updating data structures requires time consuming copying and leads to large memory consumption. Prolog programmers have developed a bag of tricks to circumvent the restriction. Pure ones based on the use of open ended data structures such as difference lists, and impure ones based on assert/retract or more efficient system specific variants of built-ins with side effects. Those tricks are not available in Mercury [15] which has no impure built-ins and whose mode system excludes the use of open ended data structures. As a consequence, the straightforward port of a Prolog application to Mercury does not always result in the anticipated speed-up [20,19]. While Mercury does provide destructive input — unique output modes, their use is cumbersome and does not fit the declarative programming paradigm where the programmer doesn't have to worry about memory management. Moreover, apart from input-output, destructive updates are not part of the current standard distribution of Mercury. The Mercury programmer has to plug in his own C-code doing the destructive updates if that is really necessary for his application [20,19]. Such practice may then conflict with optimisations done by the Mercury compiler. These conflicts can be prevented with the use of impure declarations, but in practice this is quite difficult.

Much better would be to have the compiler perform the necessary reasoning for structure reuse. A number of authors have considered this problem within single-assignment languages, in the context of logic programming languages [6,11,13], as well as functional programming languages [2,9,17,18]. Some of the approaches involve special language constructs (such as uniqueness declarations within Mercury) [1,15,21,22], others are based on compiler analyses [7,10]. Mulkers et al. [14] have developed such an analysis for Prolog, however, the lack of declarations and the impurity of Prolog make it difficult to integrate the analysis in a Prolog compiler. In [4] Bruynooghe et al. have adapted the analysis for a Mercury-like language with type, mode and determinism declarations. The current paper briefly reports on a prototype implementation of a live-structure analysis for Mercury. To achieve the long term goal of integrating the analysis in the Mercury compiler, a module based analysis is necessary. The paper develops the concepts of such an analysis where it suffices that the analysis of a module has access to the results of a goal independent analysis of the imported predicates.

Section 2 recalls the basics of the work described in [4]. Section 3 reports on the results obtained with our prototype analysis system. In section 4 module based liveness analysis is developed. We conclude with a brief discussion in Section 5.

2 Background

The goal of liveness analysis is to determine which data-structures are live at what program points. Data-structures which do not belong to the set of live structures are so-called dead, and can then be seen as possible candidates for reuse. Liveness analysis is based on the idea that within the context of a predicate, a data-structure can only be live if it will be needed during the subsequent execution of the program. More specifically, a structure is live at some program point in a predicate if it is in forward use (the structure or any of its aliases are needed by the forward execution of the program following the program point), or in backward use (i.e. the structure or any of its aliases are needed due to backtracking).

2.1 Abstract interpretation

The analysis system as presented in [4] is based on abstract interpretation [5] and uses the top-down framework of [3]. Very briefly, abstract interpretation mimics concrete execution by replacing the program's operations on concrete data with abstract operations over data descriptions. The analysis of a predicate, given abstract information about the predicate's variables (so called *call pattern*), computes abstract information for each program point, and a final abstract description of the state of the variables at exit point (*exit pattern*). For each encountered predicate call, abstract information from the caller's context is mapped unto information relevant for the called predicate (so called *procedure entry*), thus obtaining the call pattern of that predicate.

The called predicate is then analysed w.r.t. this call pattern. The obtained exit pattern will then be used to compute the abstract state of the program point following the call to this predicate in the caller's context (*procedure exit*). The analysis uses fix-point iteration to cope with recursion.

2.2 Mercury

Mercury is a logic programming language provided with types, modes and determinism declarations. The language is strongly typed and it's type system is based on a polymorphic many-sorted logic[8]. It's mode-system is such that it does not allow the use of partially instantiated structures.

Our analysis is performed at the level of the *High Level Data Structure* (HLDS) constructed by the Mercury compiler. Within this structure, predicates are *normalized*, i.e. all atoms appearing in the program have distinct variables as arguments, and all unifications $X = Y$ are explicited as either (1) a test $X == Y$, (2) an assignment $X := Y$, (3) a construction $X \Leftarrow f(Y_1, \ldots, Y_n)$, or (4) a deconstruction $X \Rightarrow f(Y_1, \ldots, Y_n)$. Within this HLDS, the atoms defining the body of a predicate are possibly reordered w.r.t. the source code and based on the mode-information: the input variables of predicate-calls must be instantiated, whereas output variables must be free.

Note that a predicate can have more then one mode declaration, yet in this paper we will assume that predicates have exactly one mode declaration [1].

2.3 Notation

As reasoning about liveness involves reasoning about data-structures, we will first introduce some definitions and notations.

Types are of particular importance to us. A type t (or if polymorphic $t(T_1, \ldots, T_n)$ with $T_1, \ldots, T_n$ type variables) is defined by one or more type constructors whose arguments are either types or type variables (only the type variables used in the type name can be used inside the constructors). It is well known that one can associate a type tree with each type.

Example 1. The polymorphic type *list*(T) is defined as:

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

Its type graph is shown in Fig. 1.

Type selectors are used to select a node in a type tree. ϵ denotes the empty selector, and t^ϵ selects the root node of t. With c_i a type-constructor, a selector s expressed as the pair (c_i, j), selects the j^{th} child of the c_i^{th} node of type t, which we write as t^s. The selected child itself can be of a type t_1. With

[1] This is no restriction of our system. If a predicate is defined with different modes, we can consider each of these modes as distinct predicates

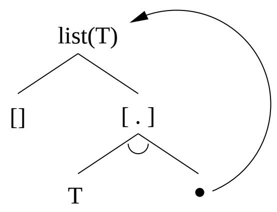

Fig. 1. Type graph of `list(T)`

s_1 a selector applicable to t_1 we have: $(t^s)^{s_1} = t^{s.s_1} = t_1^{s_1}$, with $s.s_1$ being the concatenation of selectors s and s_1. Two selectors, say s_1 and s_2 may select nodes from a type t which have the same type. In such case we say that t^{s_1} and t^{s_2} are equal. If a type t is recursive, and if the node corresponding to some selector s has type t, then t^s is simplified to t^ϵ. For example, in the context of type $list(T)$, and using "." as list constructor, $list(T)^{(.,2)}$ will be reduced to $list(T)^\epsilon$.

We define the *data-structure* X^s, where X is a variable of type t, and s is a selector for this type, as the memory cell which corresponds with the type node t^s. For our analysis, selectors will always be simplified if possible. X^ϵ is called the *top-level data-structure* of X.

Aliases are represented as pairs of data-structures: (X^{s_x}, Y^{s_y}).

2.4 Liveness analysis

With the liveness analysis of [4], we derive for each program point, the set of data-structures which are live at that point.

The *call pattern* of a predicate call $p(X_1, \ldots, X_n)$ which is to be analysised and where $X_1, \ldots, X_n$ are the so-called *head-variables* of the call, consists of a set of data-structure pairs (X^{s_x}, Y^{s_y}) expressing the possible aliases between the head-variables (GA, global aliases), and a set of data-structures relative to the head-variables which are known to be live due to the caller's context ($Live_0$). During analysis, each program point i (preceding the *current* atom) is annotated with the following abstract information:

- *local use*, LU_i: set of variables in local use which is the union of the set of variables in local forward use, LFU_i, and in local backward use, LBU_i. Variables are in local forward use if they can be accessed by the atoms following the current atom within the body of p. Variables are in local backward use if they can be accessed upon backtracking on one of the atoms in the body of p, assuming that the current atom fails.
- *local aliases*, LA_i: set of data-structure-pairs expressing which sharing is possible between the data-structures representing the values of the bound variables at the program point (and before executing the current atom).

According to [4], the set of live data-structures at program point i is then expressed as:

$$
\begin{aligned}
Live_p &= \mathcal{L}(LU_p, LA_p, GA, Live_0) && (1)\\
&= Live_0 \bigcup \{X^\epsilon | X \in LU_p\} \bigcup && (2)\\
&\quad \{X^{s_X} | (X^{s_X}, Y^{s_Y}) \in Altclos(GA, LA_p) \wedge Y \in LU_p\} \bigcup \\
&\quad \left\{ X^{s_{X_1}} \left| \begin{array}{l} (X^{s_X}, Y^{s_Y}) \in Altclos(GA, LA_p) \text{ and} \\ \exists s_1, s_2 \text{ such that } Y^{s_1} \in Live_0 \text{ and} \\ \text{either } s_Y \equiv (s_1.s_2) \wedge s_{X_1} \equiv s_X \\ \text{ or } (s_Y.s_2) \equiv s_1 \wedge s_{X_1} \equiv (s_X.s_2)\} \end{array} \right. \right\}
\end{aligned}
$$

where $Altclos(A, B)$ is the alternating closure of two sets of aliases, i.e. the set of aliases for which there exists a path alternating between elements of A and B. Intuitively, this expression states that data-structures are live if they are live due to the caller's environment – directly (first term) or indirectly (last term) – or if they are live due to their forward or backward use – again directly or indirectly, resp. second and third term.

Finally, the *exit pattern* for p will consist of the set of aliases between the head-variables after a call to p, and the set of head-variables which are in backward use through p (if for example p is a nondeterministic predicate).

Within this setting, $X^s \in Live_i$ expresses that the data-structure X^s is live, but also that for any selector s', $X^{(s.s')}$ is live too. For example, with X of type $list(T)$, we might have: (1) $X^\epsilon \in Live_i$ expressing that the list top-cell for X is live, but also all its subterms, thus that the whole list is live; (2) $X^{(\cdot,1)} \in Live_i$, $X^\epsilon \notin Live_i$, expressing that only the elements of the list are live, yet the backbone of the list is not live.

A data-structure X^s is said to be *available for reuse* at some program point i, if $X^s \notin Live_i$. There is no constraint on the children of X^s, i.e. even if, for some given s', $X^{(s.s')} \in Live_i$, yet $X^s \notin Live_i$, X^s will still be available for reuse.

Just as in [4], we check if a top-level data-structure X^ϵ becomes available for reuse at the program-point prior to the deconstruction of X ($X \Rightarrow f(X_1, \ldots, X_n)$). We say that X^ϵ can be *reused* if X^ϵ is available for reuse, and the deconstruction $X \Rightarrow f(Z_1, \ldots, X_n)$ is followed by a construction $Y \Leftarrow f(Y_1, \ldots, Y_n)$. An analysed predicate has *direct reuse* if the body of this predicate contains at least one deconstruction-construction pair for which the top-level data-structure can be reused. A predicate is said to have *indirect reuse* if it's body contains at least one call to a predicate which has direct/indirect reuse.

Liveness analysis should be seen as a phase within the compilation process, therefore the terms analysis and compilation will be used interchangeably in the remainders of the paper.

2.5 Using the analysis results

Liveness analysis indicates when a data-structure X^s become available for reuse, if at all. The most convenient way to indicate this to the compiler is to insert a pragma $reuse(X^s)$ in the HLDS at the program point following the deconstruction where the availability for reuse of X^s is decided. This pragma could then be used to provide true automatic structure reuse. The work of Taylor [16] could be adapted to make use of these pragma's instead of the tedious hand-coded destructive-input – unique output annotations. We will not handle these issues here.

3 Experimental results thus far

We have implemented a prototype of a goal-dependent liveness analysis system, covering most basic Mercury-language constructs, such that a sufficiently representative set of experiments could be performed. The goal of our experiments was to verify whether the analysis does detect reuse at the expected places, and to obtain a first idea of the computation cost. Our benchmarks consist of a set of pure academic predicates (essentially list manipulations) and a couple of real-life modules. Once we have support for modules (section 4) we plan to do more detailed experiments. Current results can be found in [12], here only a summary is given.

Reuse Detection Our experiments revealed that for most of the predicted reuses, our analysis does indeed detect reuse. Some reuses are missed due to our representation of aliases for recursive data-types as pointed out in the next paragraph.

Consider the deconstruction of a variable X of type $list(T)$: $X \Rightarrow [A|B]$. Variable B will be aliased with the tail of X, which leads to the alias: $(B^\epsilon, X^{(.,2)})$. Within our analysis this is simplified into: (B^ϵ, X^ϵ). The consequence of this simplification is that whenever B^ϵ is live, the analysis will derive that the entire structure X^ϵ is live too, although in reality only the tail of X is live, hence the possibility for a missed reuse of the top-level list-cell. A first remedy to this problem might involve a refinement of our alias-representation. However this can significantly increase the analysis cost. Another possible approach is to reorder the body of a predicate in such sense that the deconstruction is moved as closely as possible to the atom which truly uses the tail of the list, hence delaying the creation of the alias as much as possible.

Analysis cost Relating the time needed for the analysis of a module, T^a, with the time needed for compiling this module without analysis, T^c, we obtained that in average $T^a/T^c \sim 0.25$. Taking into account that while efficiency was a concern in the design of our prototype, it was sometimes sacrificed in favor of development ease and extendibility, this average relative cost seems acceptable. Yet, our experiments also revealed some cases for which the relative cost was not acceptable at all ($T^a/T^c \sim 10$). This high cost is mainly related to the complexity of the types used, as the following example will illustrate. Consider a type t defining n functors, which all have the same arity $m \geq 1$, and arguments of some same other type, then the number of possible selectors will be equal to $n \cdot m$. The complexity of the alternative closure operation is known to be exponential to this number of selectors. With increasing size of n and m, it is evident that the global computation cost becomes unbearable. Such situations should therefore be avoided as much as possible. A possible approach might consist in artificially reducing the number of selectors[2], a typical widening operation. This widening will induce possible loss of precision. Future work has to determine what the best tradeoff between cost and precision will be.

Yet, the main problem with the current prototype is it's lack of support for modules, which is the subject of the second and main part of this paper.

4 Towards a module based analysis

Modern programming languages allow large applications to be distributed over several modules, allowing separate compilation of these. For the compilation of one module, only a small amount of information about the imported modules is needed. This information is generated during the compilation of the latter modules, and is typically stored in a separate file. This is also the model followed by Mercury.

While the goal-dependent liveness analysis system of sections 2 and 3 yields positive results, it does not match with this model though, as in order to fully optimize a predicate and the predicates it depends on, the full source code is needed. It would also require reanalysis and possibly recompilation of all the imported modules. Although the resulting compiled code will be highly optimized, the cost of these constant recompilations is unbearable.

Essentially module based analysis can be split into two subproblems: intra-module optimization —safely analyse a given predicate with minimal information about imported predicates— and inter-module optimization — making safe decisions on whether a predicate can use an optimized version of an imported predicate or not. In section 4.1 we discuss intra-module optimization, so-called weak module support. Section 4.2 introduces inter-module optimization (strong module support), where also the concept of goal-independent liveness analysis is defined. Finally, section 4.3 combines strong and weak module support into full module support.

[2] e.g. by designating all arguments of a functor at once by a unique selector, thus in a sense treating all arguments in the same way.

4.1 Weak module support

Consider a predicate t of which the body contains a call to q, and where t and q are defined in different modules. q is said to be an *external* or *imported* predicate w.r.t. the module in which t is defined.

Let i be the program point in t before the call to q, and $i+1$ the program point after the call to the external predicate. Then the set of live variables at these program points can be expressed as:

$$\begin{aligned} Live_i^t &= \mathcal{L}(LU_i^t, LA_i^t, GA^t, Live_0^t) \\ Live_{i+1}^t &= \mathcal{L}(LU_{i+1}^t, LA_{i+1}^t, GA^t, Live_0^t) \end{aligned}$$

As argued in [4], only the local uses and aliases (LU^t, LA^t) are program-point dependent.

Let's first consider LU_{i+1}^t for which we have: $LU_{i+1}^t = LFU_{i+1}^t \cup LBU_{i+1}^t$. The forward use component, LFU_{i+1}^t, is independent of q, as it simply contains those variables which are still used after this program point. On the other hand, the backward use component is not. Typically, if q is a non-deterministic predicate, then it will introduce additional variables in local backward use. Yet, whether q introduces these additional variables or not is totally independent of the variables which are already in backward use, hence we can express LBU_{i+1}^t as $LBU_{i+1}^t(LBU_i^t, LBU^q, \ldots)$, where LBU^q is the set of variables in local backward use due to q. The latter can be computed independently.

The local aliases can similarly be expressed in terms of the already existing aliases, and those due to the external predicate. As stated in [4]: $LA_{i+1}^t = Altclos(LA_i^t, LA^q)$, i.e. the set of local aliases in a program point can be approximated by the alternating closure between the already existing local aliases and the set of additional aliases created by the preceding call. Again, q's contribution is totally independent of the already existing aliases, and can therefore be derived independently.

In summary, in order to correctly analyse predicate t, the only information needed about the external predicate q is: LBU^q and LA^q. This information is independent of any specific call-pattern, and can therefore be derived during compilation of the module to which q belongs (either by a dedicated analysis, or as a result of a goal-independent liveness analysis as will be mentioned later).

Note that here we are only interested in trying to optimize t, but not the external predicates, hence the term *weak* module support.

4.2 Strong module support

Consider again a predicate t which calls a predicate q, both being defined in different modules. Now suppose that a goal-dependent liveness analysis of q under some artificial initial abstract substitution, would reveal possible reuse within q. We could then create multiple versions of q: one basic version without reuse, and a number of different versions of q exploiting each form of detected reuse[3]. For each of these reuse-versions, we would have to express conditions, so called *reuse conditions*, which would have to be verified by the caller in order to safely decide for a version of q with reuse or not. These conditions could then be saved into a separate file, serving as interface for the module to which q belongs, and avoiding herewith recompilation of that module each time it is imported into another module.

To achieve this, two questions must be answered. How is a module-predicate, say q, to be analysed in order to derive maximal information given minimal knowledge about the possible call patterns? This will lead us to the concept of goal-independent liveness analysis. And how can we express conditions for reuse? These conditions must be easy to derive, and to verify.

4.2.1 Goal-independent analysis A goal-dependent analysis of a predicate consists of analysing that predicate, given it's initial call pattern. This call pattern consists of a set of data-structures related to the head variables which are known to be live anyway ($Live_0$), and a set of aliases which might exist between the arguments with which the predicate is called (GA). Let R_1 be the number of opportunities for reuse detected in this setting.

Consider another analysis of the same predicate, under the assumption that no variables are known to be a priori live ($Live_0 = \emptyset$), and no aliases exist between the arguments ($GA = \emptyset$). In such a setting structures will only be live depending on their local use. If $Live_0$ and GA are not empty, then this will always result in bigger live-sets. Therefore it is obvious that the analysis will detect the maximal set of possible reuses, let R_{max} be the size of this set. We have: $R_{max} \geq R_1$. Yet, in this setting we risk to detect opportunities for reuse which are unrealistic and known to be seldom applicable, resulting in extra versions of the predicate of which the usability is known to be small. Indeed, Mercury is a moded language: every argument of a call is either input or output. While examples can be found where even output variables might become candidates for reuse[4], it is realistic to assume that the data-structures corresponding to the output variables are live within the context of the caller.

This leads us to a third possible analysis of the predicate, for which $Live_0$ consists of the top-cell data-structures of the output-arguments, and where $GA = \emptyset$. Let R_2 be the number of possible opportunities for reuse. We have: $R_{max} \geq R_2 \geq R_1$. Here R_2 will reflect the maximal set of realistic reuses. We define this analysis setting as the *goal-independent liveness analysis* of the considered predicate, as it is the most general practical liveness analysis possible, and although the analysis is in fact a goal-dependent analysis (Sect. 2),

[3] Theoretically, if n opportunities for reuse are detected, 2^n different versions for q can be provided. See section 4.2.3 for practical issues on this matter.

[4] e.g. a predicate with two output arguments X and Y. In a first step X is constructed, in a second step Y is constructed based on X. If X is not used within the context of the caller, then Y could be constructed reusing data cells of X.

the used call-pattern is fully independent of a true global goal-dependent analysis one might perform.

In next section we derive what extra information needs to be gathered during this goal-independent analysis for expressing reuse conditions.

4.2.2 Expressing conditions for reuse In what follows, a component is given a subscript i when its value depends on the program point i and it is given a superscript gi or gd when its value differs between the goal independent and the goal dependent analysis.

4.2.2.1 Direct reuse Let q be a predicate for which a goal-independent analysis has been performed, and for which exactly one opportunity for reuse has been detected: a variable, say X is deconstructed, it's top-level X^ϵ becomes dead and can be reused in some following construction (direct reuse). Let i be the program-point just before the deconstruction, and $Live_i^{gi}$ the live set at that program-point. For a goal-independent case, we have:

$$Live_i^{gi} = \mathcal{L}(LU_i, LA_i, \emptyset, Live_0^{gi}) \tag{3}$$

where $Live_0^{gi}$ solely comprises the output arguments of q. Note that LU and LA are independent of the call pattern.

As reuse is detected we must have that $X^\epsilon \notin Live_i^{gi}$.

Consider the call pattern for q during a goal-dependent analysis of some other predicate. The corresponding analysis obtains:

$$Live_i^{gd} = \mathcal{L}(LU_i, LA_i, GA^{gd}, Live_0^{gd}) \tag{4}$$

Reuse is allowed if and only if $X^\epsilon \notin Live_i^{gd}$.

Expressions 3 and 4 differ only in their global components, so a brute force approach to verify for reuse could be as follows. At the end of the goal-independent analysis, we simply save the local components LU_i and LA_i. When during a goal-dependent analysis q is called, LU_i and LA_i of q can be used to compute $Live_i^{gd}$ with (2) (together with GA^{gd} and $Live_0^{gd}$ from the calling context). The reuse-version of q can be used if X^ϵ does not belong to $Live_i^{gd}$. Although conceptually very easy, this method has certain drawbacks. The body of q may contain many local variables, LU_i and LA_i may then be relatively large sets. Computing $Live_i^{gd}$ can become rather expensive. Therefore we must examine whether the amount of information to be saved can be reduced, as well as the cost of verifying reuse.

Comparing the explicited formulas (2) for $Live_i^{gi}$ and $Live_i^{gd}$, and given that $X^\epsilon \notin Live_i^{gi}$, we can observe that $X^\epsilon \notin Live_i^{gd}$ only if the following is true:

1. $X^\epsilon \notin Live_0^{gd}$
2. $\nexists Y : (X^\epsilon, Y^s) \in Altclos(GA^{gd}, LA_i) \wedge Y \in LU_i$

3. $\nexists Y : (X^\epsilon, Y^{s_Y \cdot s}) \notin Altclos(GA^{gd}, LA_i) \wedge Y^{s_Y} \in Live_0^{gd}$

We will now examine each of these conditions.

Condition 1. To check condition 1 during the goal-dependent analysis it suffices to know the name of the data-structure which might be reused. During that analysis one simply needs to perform the procedure-entry operation, thus obtaining $Live_0^{gd}$, and verify whether the concerned data-structure belongs to this set or not.

Condition 2. We will first start with some lemma's and definitions. Selectors which are irrelevant for the discussion are omitted.

Let $\mathcal{H}_{in}$ be the set of input head-variables of the external predicate q. Let $var(E)$ denote the set of variables in the expression E[5].

Lemma 1. $var(GA^{gd}) \subseteq \mathcal{H}_{in}$.

By definition, GA^{gd} relates only to head-variables. Due to Mercury's moded nature, output variables are known to be free variables at procedure-entry, hence no aliases with these can exist at that moment.

Lemma 2. $\nexists \alpha, \beta : (\alpha, \beta) \in LA_i \wedge \alpha, \beta \in \mathcal{H}_{in}$.

Mercury does not allow partially instantiated variables to be passed around, hence no new aliases between input variables can be created by the called procedure.

Recall that given two set of aliases A and B, $Altclos(A, B)$ will consist of aliases (α, β) for which there exists a path (with length ≥ 1) of aliases alternating between elements of A and B (see [4]).

Definition 41 *Given sets of aliases A and B, $Altclos_i(A, B)$ is the set of aliases for which there exists a path of length i alternating between aliases of A and B.*

Note that $Altclos_1(A, B) = A \cup B$.

Example 2. Let $A = \{(a, b), (c, d)\}$, and $B = \{(a, c), (d, e)\}$. To compute $Altclos_1(A, B)$, one needs to construct only paths of length 1, therefore $Altclos_1(A, B) = A \cup B$. The only paths of length 2 are: $(b, a) - (a, c), (d, c) - (c, a), (c, d) - (d, e)$. Therefore $Altclos_2(A, B) = \{(b, c), (d, a), (c, e)\}$. Paths of length 3: $(b, a) - (a, c) - (c, d), (e, d) - (d, c) - (c, a)$, thus $Altclos_3(A, B) = \{(b, d), (e, a)\}$. Finally the only path of lenght 4 is: $(b, a) - (a, c) - (c, d) - (d, e)$ and $Altclos_4(A, B) = \{(b, e)\}$.

Definition 42 *Given sets of aliases A and B, $Altclos_{\geq i}(A, B)$ is the set of aliases for which there exists a path of length $\geq i$ alternating between aliases of both sets.*

[5] If E is a variable with a selector, say X^{s_x}, we will use the notation: $E \in Set$, instead of $var(E) \subseteq Set$, where Set represents some set of variables

Note that $Altclos_{\geq i}(A,B) = Altclos_i(A,B) \cup Altclos_{\geq i+1}(A,B)$.

Lemma 3. *The paths generated for the computation of* $Altclos_3(GA^{gd}, LA_i)$ *will have the shape* $L_1 - G - L_2$*, where* $L_{\{1,2\}} \in LA_i$ *and* $G \in GA^{gd}$.

Suppose a path $G_1 - L - G_2$, where $G_{\{1,2\}} \in GA^{gd}$ and $L \in LA_i$, would have been generated for $Altclos_3(GA^{gd}, LA_i)$. Given that $var(GA^{gd}) \subseteq \mathcal{H}_{in}$ (lemma 1), such path would imply: $var(L) \subseteq \mathcal{H}_{in}$, which contradicts lemma 2.

Lemma 4. $Altclos_{\geq 4}(GA^{gd}, LA_i) = \emptyset$.

Indeed, each alternating path of length ≥ 4 will have to contain a subpath of shape $G_1 - L - G_2$, with $G_{\{1,2\}} \in GA^{gd}$ and $L \in LA_i$, yet this was shown to be impossible.

Lemma 5. *Let* $LA_i|_{\mathcal{H}_{in}}$ *be the subset of aliases* (α, β) *of* LA_i *for which either* α *or* β *belongs to* $\mathcal{H}_{in}$. $Altclos(GA^{gd}, LA_i) = Altclos(GA^{gd}, LA_i|_{\mathcal{H}_{in}})$.

This is again a direct consequence of the first two lemma's.

Using these lemma's and definitions, we can reformulate and split condition 2 for reuse as:

$$\nexists Y : (X^{\epsilon}, Y^{s}) \in GA^{gd} \wedge Y \in LU_i \tag{5}$$

$$\nexists Y : (X^{\epsilon}, Y^{s}) \in LA_i \wedge Y \in LU_i \tag{6}$$

$$\nexists Y : (X^{\epsilon}, Y^{s}) \in Altclos_2(GA^{gd}, LA_i|_{\mathcal{H}_{in}}) \wedge Y \in LU_i \tag{7}$$

$$\nexists Y : (X^{\epsilon}, Y^{s}) \in Altclos_3(GA^{gd}, LA_i|_{\mathcal{H}_{in}}) \wedge Y \in LU_i \tag{8}$$

Note that $var(GA^{gd}) \subseteq \mathcal{H}_{in}$, therefore we can limit the verification of (5) for all Y belonging to $LU_i|_{\mathcal{H}_{in}}$, i.e. the subset of LU_i related to input head-variables only.

Expression (6) is always satisfied. Indeed, suppose that there would be such a $Y \in LU_i$ for which $(X^{\epsilon}, Y^{s}) \in LA_i$, then according to (2) for $Live_i^{gi}$, we would have $X^{\epsilon} \in Live_i^{gi}$, which contradicts our starting point.

Expression (7) is equivalent to the statement: $\nexists \beta, Y : (X^{\epsilon}, \beta) - (\beta, Y^{s}) \in$ set of paths formed in $Altclos_2(GA^{gd}, LA_i|_{\mathcal{H}_{in}})$ and $Y \in LU_i$. This condition can be split in two parts: (X^{ϵ}, β) either belongs to GA^{gd} or $LA_i|_{\mathcal{H}_{in}}$:

- $\nexists \beta : (X^{\epsilon}, \beta) \in GA^{gd} \wedge (\beta, Y^{s}) \in LA_i|_{\mathcal{H}_{in}} \wedge Y \in LU_i$. The third component in (2) for $Live_i^{gi}$ is exactly $\{\beta | (\beta, Y^{s}) \in LA_i|_{\mathcal{H}_{in}} \wedge Y \in LU_i\}$, which we denote as $Live3_i^{gi}$. Note that as GA^{gd} relates to input variables, we can limit β by requiring it to belong to $\mathcal{H}_{in}$. Hence, with $Live3_i^{gi}|_{\mathcal{H}_{in}}$ defined as the set of input head-variables belonging to $Live3_i^{gi}$, we obtain: $\nexists \beta : (X^{\epsilon}, \beta) \in GA^{gd} \wedge \beta \in Live3_i^{gi}|_{\mathcal{H}_{in}}$.
- $\nexists \beta : (X^{\epsilon}, \beta) \in LA_i|_{\mathcal{H}_{in}} \wedge (\beta, Y^{s}) \in GA^{gd} \wedge Y \in LU_i$. According to lemma 1, $Y \in \mathcal{H}_{in}$. Here we are only interested in local aliases related to X^{ϵ}, we will denote this set as $LA_i|_{\mathcal{H}_{in}, X^{\epsilon}}$. We obtain: $\nexists \beta : (X^{\epsilon}, \beta) \in LA_i|_{\mathcal{H}_{in}, X^{\epsilon}} \wedge (\beta, Y^{s}) \in GA^{gd} \wedge Y \in LU_i|_{\mathcal{H}_{in}}$.

Using lemma 3, (8) is equivalent to: $\not\exists \beta, \gamma, Y : (X^\epsilon, \beta) \in LA_i|_{\mathcal{H}_{in}} \wedge (\beta, \gamma) \in GA^{gd} \wedge (\gamma, Y^s) \in LA_i|_{\mathcal{H}_{in}} \wedge Y \in LU_i$. The last two terms imply that $\gamma \in Live3_i^{gi}$. According to lemma 1, we also have that $\{\gamma, \beta\} \subseteq \mathcal{H}_{in}$. We can further limit β by observing that if $\beta \in LU_i$, then $X^\epsilon \in Live_i^{gi}$ (formula (2)), which contradicts our starting point. Again we are only interested in the set of local variables concerning X^ϵ. Hence we obtain: $\not\exists \beta, \gamma : \beta \in (\mathcal{H}_{in} \setminus LU_i|_{\mathcal{H}_{in}}) \wedge \gamma \in Live3_i^{gi}|_{\mathcal{H}_{in}} \wedge (X^\epsilon, \beta) \in LA_i|_{\mathcal{H}_{in}, X^\epsilon} \wedge (\beta, \gamma) \in GA^{gd}$.

Condition 3. Using a very similar reasoning as for condition 2, we can derive that condition 3 splits up into three parts, see table 1.

Summary. Condition 1 resulted in one expression to be verified, condition 2 yielded four checks to be made, condition 3 added again three verifications, this brings us to a total of eight expressions to be verified. Table 1 summarizes them all. The information to be saved during the goal-independent analysis is reduced to the name of the variable which can be reused (X^ϵ), as well as the following sets: $LU_i|_{\mathcal{H}_{in}}$, $Live3_i^{gi}|_{\mathcal{H}_{in}}$ and $LA_i|_{\mathcal{H}_{in}, X^\epsilon}$. Note that from these sets all information regarding local variables has been filtered out (except for X^ϵ). The verifications are simple projections of sets, hence they will be cheap to verify.

Yet having to verify eight conditions each time appears as a high cost to pay. We can observe that if $X \in \mathcal{H}_{in}$ then $LA_i|_{\mathcal{H}_{in}, X^\epsilon} = \emptyset$, and all conditions related to this set will always be true, resulting in only four conditions to be verified. On the other hand, if $X \notin \mathcal{H}_{in}$, then there will never exist a β such that $(X^\epsilon, \beta) \in GA^{gd}$, which eliminates the conditions depending on this relation. We will also have $X^\epsilon \notin Live_0^{gd}$. This also results in only four conditions to be met. Therefore, practically, we will never have to verify explicitly all eight conditions, as only four of them will have to be verified each time, the others being automatically fulfilled depending on whether the reusable structure is a head-variable or not. This is also summarized in table 1.

Note that due to the accuracy of the derivation of the conditions, verifying these small conditions, or verifying whether the reusable data-structure belongs to $Live_i^{gd}$ by computing the latter from scratch will, though with different computation costs, yield exactly the same results, hence no loss of precision is introduced at this level

4.2.2.2 Indirect reuse Consider a predicate q_1 for which a goal-independent analysis has been performed. Suppose this analysis detected indirect reuse with respect to some predicate q_2. Let $X_{q_2}^\epsilon$ be the data-structure which q_2 claims to be reusable. A possible strategy for defining conditions of reuse in terms of q_1 could be to translate the data-structure $X_{q_2}^\epsilon$ in terms of the variables with which q_2 has been called, and express similar conditions as above in terms of these variables. This translation can be based on the aliasing

$X \in \mathcal{H}_{in}$	$X^{\epsilon} \notin Live_0^{gd}$	1
	$\nexists Y : (X^{\epsilon}, Y^{s}) \in GA^{gd} \wedge Y \in LU_i\vert_{\mathcal{H}_{in}}$	2
	$\nexists \beta : (X^{\epsilon}, \beta) \in GA^{gd} \wedge \beta \in Live3_i^{gi}\vert_{\mathcal{H}_{in}}$	2
	$\nexists Y : (X^{\epsilon}, Y^{s_y \cdot s}) \in GA^{gd} \wedge Y^{s_y} \in Live_0^{gd}$	3
$X \notin \mathcal{H}_{in}$	$\nexists \beta, Y : (X^{\epsilon}, \beta) \in LA_i\vert_{\mathcal{H}_{in}, X^{\epsilon}} \wedge (\beta, Y^{s}) \in GA^{gd}$ $\wedge Y \in LU_i\vert_{\mathcal{H}_{in}}$	2
	$\nexists \beta, \gamma, Y : (X^{\epsilon}, \beta) \in LA_i\vert_{\mathcal{H}_{in}, X^{\epsilon}} \wedge (\beta, \gamma) \in GA^{gd}$ $\wedge \beta \in (\mathcal{H}_{in} \setminus LU_i\vert_{\mathcal{H}_{in}}) \wedge \gamma \in Live3_i^{gi}\vert_{\mathcal{H}_{in}}$	2
	$\nexists Y : (X^{\epsilon}, Y^{s_y \cdot s}) \in LA_i\vert_{\mathcal{H}_{in}, X^{\epsilon}} \wedge Y^{s_y} \in Live_0^{gd}$	3
	$\nexists \beta, Y : (X^{\epsilon}, \beta) \in LA_i\vert_{\mathcal{H}_{in}, X^{\epsilon}} \wedge (\beta, Y^{s_y \cdot s}) \in GA^{gd}$ $\wedge Y^{s_y} \in Live_0^{gd}$	3

Table 1. Summary of the expressions to be verified in order to safely decide for using the predicate version which reuses X^{ϵ} or not. The last column refers to the condition (1, 2 or 3) of which the expression has been derived.

information between X_{q_2} and the head-variables of q_2. Further work on this part of reuse-verification is required.

4.2.3 Practical issues A goal-independent analysis of a predicate might reveal more than one opportunity for reuse. Each of these opportunities corresponds with a different set of conditions to be fulfilled by the caller. Now, if we want a compile-time garbage collection system which truly exploits every possible form of data-reuse, and suppose a goal-independent analysis of some predicate reveals n opportunities, then we would have to generate 2^n different versions, resulting in a real code-explosion. Therefore, in the implementation of real analysis systems, a tradeoff will have to be made between the size of the compiled code and the number of reuses achieved. A possible strategy could consist of only creating two versions of such a predicate: a first version without reuse, and a second version with every possible reuse foreseen. The conditions which have to be fulfilled by the caller of this predicate will become more severe, risking that reuse is only possible in a few cases. Future work will have to determine which strategies are feasible.

Another issue which has not been mentioned yet is the problem of mutually recursive modules. Although the theory developed in previous paragraphs is independent of the module-dependencies which might exist, practically speaking, mutually recursive modules will be a problem. Whatever strategy one will use to handle such cases, it will always induce a certain loss of precision.

4.3 Full module support

Weak module support is possible if LBU and LA is available for each exported predicate of a Mercury-module. As said, this information can be obtained by a dedicated analysis.

Strong module support consists of performing goal-independent analyses of all exported predicates of the modules used. Such analyses yield information on whether reuse is possible at all, and if so, provide the reuse conditions to which eventual caller's will have to comply in order to allow the use of the reuse-version of the predicates. During a goal-independent analysis of a predicate, the sets LBU and LA are being computed anyway, therefore no special analysis has to be foreseen to deduce these sets: it can all be computed during goal-independent liveness analysis.

While weak module support allows the detection of possible reuses within the body of a predicate using external predicates, strong module support also enables us to safely decide whether it is allowed or not to use a reuse-version of the used external predicates. Weak and strong module support are therefore complementary. Combining both we obtain a full modular analysis.

5 Conclusion

We have implemented a prototype system for goal-dependent liveness analysis of Mercury. Results obtained with this prototype have been very positive (precision as well as analysis cost), yet revealed two possible problems. First of all, some potential reuses are missed due to our representation of recursive data-structures. In the presence of complicated type-definitions, a second problem might occur, as the analysis risks to become exponential. We have briefly mentioned possible solutions to both problems, suggesting that further optimization of basic goal-dependent liveness analysis will have to be done. Generally, a tradeoff will always have to be made between analysis cost and precision. Although the prototype already covers basic Mercury language constructs, it must be extended to cover them all (such as higher-order predicates and type-classes which are not yet supported).

Even in the presence of a full optimal goal-dependent liveness analysis, the potential of reuse-detection can only be fully exploited if support for modular analysis is provided. In this paper, we introduced the concept of weak modular support, which allows to analyse a predicate in a goal-dependent way, even in the presence of external predicates. We also defined the notions of strong module support and goal-independent liveness analysis, such that when analysing a predicate which calls an external predicate, we can safely decide whether this predicate may use a reuse-version of this external predicate or not. The information needed from the goal-independent analysis of the latter, as well as the cost of making this decision have been reduced by deriving clear-cut conditions for reuse. In the case of direct reuses, expressing and verifying these conditions introduces no loss of precision. This might be different for indirect reuses though, and has to be further investigated. Further work should also show an optimal strategy for keeping the number of reuse-versions of a predicate to a realistic level.

Our long term goal is to incorporate a full compile-time garbage collection system within the Mercury compiler. This paper is already one step closer towards such an ecological Mercury.

References

1. Yves Bekkers and Paul Tarau. Monadic constructs for logic programming. In John Lloyd, editor, *Proceedings of the International Symposium on Logic Programming*, pages 51–65, Cambridge, December 4–7 1995. MIT Press.
2. A. Bloss. Path analysis and the optimization of non-strict functional languages. Technical Report YALEU/DCS/RR-704, Department of Computer Science, Yale University, New Haven, CT, 1988.
3. Maurice Bruynooghe. A practical framework for the abstract interpretation of logic programs. *Journal of Logic Programming*, 10(2):91–124, February 1991.
4. Maurice Bruynooghe, Gerda Janssens, and Andreas Kågedal. Live-structure analysis for logic programming languages with declarations. In L. Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming (ICLP'97)*, pages 33–47, Leuven, Belgium, 1997. MIT Press.
5. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, 1977.
6. Saumya K. Debray. On copy avoidance in single assignment languages. In David S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*, pages 393–407, Budapest, Hungary, 1993. The MIT Press.
7. G. Gudjonsson and W. Winsborough. Update in place: Overview of the Siva project. In D. Miller, editor, *Proceedings of the International Logic Programming Symposium*, pages 94–113, Vancouver, Canada, 1993. The MIT Press.
8. Fergus Henderson, Thomas Conway, Somogyi Zoltan, and Jeffery David. The mercury language reference manual. Technical Report 96/10, Dept. of Computer Science, University of Melbourne, February 1996.
9. S. B. Jones and D. Le Métayer. Compile-time garbage collection by sharing analysis. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture '89, Imperial College, London*, pages 54–74, New York, NY, 1989. ACM.
10. Andreas Kågedal and Saumya Debray. A practical approach to structure reuse of arrays in single assignment languages. In Lee Naish, editor, *Proceedings of the 14th International Conference on Logic Programming*, pages 18–32, Cambridge, July 8–11 1997. MIT Press.
11. Feliks Kluźniak. Compile-time garbage collection for ground Prolog. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 1490–1505, Seattle, 1988. MIT Press, Cambridge.

12. N. Mazur, G. Janssens, and M. Bruynooghe. Towards memory reuse for Mercury. Report CW278, Department of Computer Science, Katholieke Universiteit Leuv en, June 1999. http://www.cs.kuleuven.ac.be/publicaties/rapporten/CW1999.html.
13. Anne Mulkers, Will Winsborough, and Maurice Bruynooghe. Analysis of shared data structures for compile-time garbage collection in logic programs. In D. H. D. Warren and P. Szeredi, editors, *Proceedings of the Seventh International Conference on Logic Programming*, pages 747–762, Jerusalem, 1990. MIT Press, Cambridge.
14. Anne Mulkers, Will Winsborough, and Maurice Bruynooghe. Live-structure dataflow analysis for Prolog. *ACM Transactions on Programming Languages and Systems*, 16(2):205–258, March 1994.
15. Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *The Journal of Logic Programming*, 29(1–3):17–64, October-December 1996.
16. Simon Taylor. Optimization of Mercury programs. Honours report, Department of Computer Science, University of Melbourne, November 1998.
17. Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen Højfeld, Peter Sestoft, and Peter Bertelsen. Programming with regions in the ML Kit. Technical Report D-342, Dept. of Computer Science, University of Copenhagen, 1997.
18. Mads Tofte and Talpin Jean-Pierre. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.
19. H. Vandecasteele. *Constraint Logic Programming: Applications and Implementation*. PhD thesis, Department of Computer Science, K.U. Leuven, May 1999.
20. H. Vandecasteele, B. Demoen, and J. Van Der Auwera. The use of Mercury for the implementation of a finite domain solver. In I. de Castro Dutra, M. Carro, V. Santos Costa, G. Gupta, E. Pontellia, and Silva F, editors, *Nova Science Special Volume on Parallelism and Implementation of Logic and Constraint Logic Programming*. Nova Science Publishers Inc, 1999.
21. Philip Wadler. The essence of functional programming. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, Albequerque, New Mexico, January 1992.
22. Philip Wadler. How to declare an imperative (invited talk). In *International Logic Programming Symposium*, Portland, Oregon, December 1995. MIT Press.