A Module Based Analysis for Memory Reuse in Mercury*

Nancy Mazur, Gerda Janssens, and Maurice Bruynooghe

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

Abstract. In previous work Bruynooghe, Janssens and Kågedal developed a live-structure analysis for Mercury which detects memory cells available for reuse. Separate compilation of modules is an essential ingredient of a language such as Mercury which supports programming in the large. Hence, to be practical, a live-structure analysis also has to be module based. This paper develops a modular live-structure analysis and extends it with a modular reuse analysis. It also describes preliminary results obtained with a first prototype of the module based analysis.

1 Introduction

In declarative languages, the programmer is liberated from the low level details of memory management such as allocation of memory and destructive updates of data structures. The price to pay for this convenience is a loss of performance due to the run-time overhead of garbage collection, due to an increased number of cache misses (caused by the loss of locality of data structures) and due to the overhead of creating new data structures (creating a new version of a data structure is typically more expensive than updating an existing one).

There has been a lot of research on methods to overcome this handicap and improve the memory management, both for logic programming languages [5, 12, 14], as for functional programming languages [10, 19]. Some approaches depend on a combination of special language constructs and analysis [17, 1, 21], others are solely based on compiler analyses [7, 11, 20]. At least in logic programming, none of the analysis based methods has reached the maturity of becoming part of a widely distributed implementation, so this largely remains an unsolved problem.

A language as Mercury, a logic language with declarations [17], offers a partial solution through the availability of destructive input *di* and unique output *uo* modes; however the use of these modes is cumbersome for the programmer. Moreover it does not fit into the declarative programming paradigm. Hence we believe it is a useful research goal to develop a reuse analysis for Mercury. In addition, mastering it for Mercury should be a useful stepping stone for developing such an analysis for systems such as Ciao Prolog [9] where declarations are

* This work has been supported by the GOA project LP^+, the ESPRIT project ARGo, and the FWO-Vlaanderen.

optional and where one has to cope with the impurities of Prolog and HAL [6] which is a constraint language having many similarities with Mercury.

Mulkers et al. [14] have developed an analysis for Prolog which detects when memory cells become available for reuse; however the lack of declarations and the impurity of Prolog make it rather infeasible to obtain acceptable analysis times and to integrate it in a Prolog compiler. In [3] Bruynooghe et al. have adapted the analysis for a Mercury-like language with type, mode and determinism declarations; the analysis takes also backtracking into account (the original work relied on the trail to restore overwritten data structures on backtracking). The paper only briefly sketches some preliminary ideas about how to make the analysis modular. Modularity is essential for the analysis to be practical as it is infeasible to analyse large programs (e.g. the Mercury compiler) as a single unit. The concept of modules is an integral part of the Mercury support for *programming in the large* and even simple applications import predicates from libraries. Moreover, Mercury compiles each module of a program separately, hence an analysis which has to become part of the compiler better does the same. This paper presents a module based analysis for memory reuse as a two step analysis where a so called default liveness analysis of the procedures exported by the module is followed by a reuse analysis. Results obtained by a prototype are presented. Finally the paper suggests also some solutions w.r.t. the creation of multiple versions for procedures.

The paper is a reworking and elaboration of [13] where the non modular default analysis was first described. Modularisation of analysis is also a current research issue in the Ciao Prolog project. A discussion of possible approaches which have quite some parallels with our work is in [15].

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

2 Background

2.1 Abstract interpretation

The analysis system in [3] uses the top-down abstract interpretation framework of [2]. 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 arguments (*call pattern*), computes abstract information for each program point, and a final abstract description of the state of the variables at the exit point (*exit pattern*). For each predicate call, abstract information from the caller's context is mapped onto information relevant for the called predicate (*procedure entry*), thus obtaining the call pattern of that predicate. The called predicate is analysed w.r.t. this call pattern. The obtained exit pattern will be used to compute the abstract state of the program point following the predicate call (*procedure exit*). The analysis uses fixpoint iteration to cope with recursion.

2.2 Mercury

Mercury [8] is a logic programming language with types, modes and determinism declarations. Its type system is based on a polymorphic many-sorted logic and its mode-system 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 one of (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 the HLDS, the atoms of a clause body are –if needed– reordered such that the body is well moded.

Just like in the HLDS we will use the notion of a *procedure*, i.e. a combination of one predicate with *one* mode, and thus talk about the analysis of a procedure.

2.3 Types and selectors

Using a simple example, we recall some basics about types.

The polymorphic type $list(T)$ is defined as: `list(T)--->[] ; [T|list(T)]`. This type is obtained by applying a *type constructor* (`list`) on zero or more types or type variables (here type variable `T`). Its definition is given by the right hand side of the above expression and consists of one or more alternatives. Each alternative is a distinct *type functor* applied to zero or more types or type variables (`[]` has zero arguments, the list constructor has two, namely `T` and `list(T)`). A *type tree* is a possibly infinite graphical representation of a type definition. A finite *type graph* is obtained by imposing that two type nodes on the same branch from the root are the same when they are labelled with the same type. The graph of `list(T)` is shown in Fig. 1: it has two type nodes (`list(T)` and `T`) and two functor nodes (`[]` and `[.]`).

Type selectors are used to select type nodes in type graphs. The empty selector is denoted by ϵ; with t a type, t^ϵ selects the root node of its type graph. If t_0^s selects a node of type t_1 in the type graph of t_0 and one of the alternatives defining t_1 has a type functor f/n then $t_0^{s.(f,j)}$ with $j \leq n$ selects the type node which is the j^{th} child of the functor node labeled f/n that is a child of t_1.

With recursive types, several selectors can select the same node in a type graph. This equivalence is denoted with $s_1 \equiv s_2$, with s_1 and s_2 selectors. Using "." as list constructor, $list(T)^{(.,2)}$ and $list(T)^\epsilon$ both select the root node. Our analysis always simplifies selectors, hence will replace the former by the latter.

Terms also have a tree representation and nodes of the term tree can be mapped to type nodes of the corresponding type tree. With X a variable which has a term of type t as value and with s a selector for t, X^s denotes the nodes in the term tree of X which are mapped to t^s. We refer to the memory cells implementing these nodes as the *data structure* of X^s. X^ϵ selects (at least) the root node of the term tree. This root node is the *top-level data structure* of X.

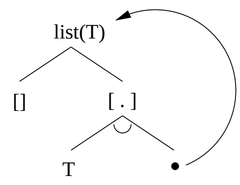

Fig. 1. Type graph of `list(T)`

2.4 Liveness analysis

Our liveness analysis exploits the mode, type and determinism information available in the HLDS of Mercury programs. It can be applied on any sequential logic programming language where this information is available through declarations or analysis. Due to the absence of partially instantiated data structures every unification can be reduced to one of the cases in Section 2.2. Although the analysis can handle full unification, the ultimate reuse will depend on the presence of deconstruction/construction pairs. With less precise modes, less of these pairs occur in the code.

The liveness analysis of [3] computes in each program point which data structures are live. Calling the program point under consideration the *current* program point and the atom following it the *current* atom, a data structure is **live** in the *current* program point if –given the bindings of the program variables before executing the current atom– the data structure is reachable

- either from a program variable occurring in an atom following the *next* program point (hence it will possibly be accessed while continuing the execution after successful completion of the current atom).
- or from a program variable which will possibly be accessed after the backtracking which would occur if the execution of the current atom fails.

Liveness is expressed as a set of elements X^s. That X^s is live in program point p means that the data structure(s) selected by X^s in the current value of the variable X will probably be accessed after executing the current atom.

The liveness set in a program point p is computed from two components:

- A binary relation *ALIAS*. Tuples in this relation are of the form (X^s, Y^t). The intuitive meaning is: if data structure X^s is live then Y^t is *possibly* live and, if Y^t is live, then X^s is *possibly* live[1]. The relation is not transitive as we maintain only one *ALIAS* relation in each program point, hence it is possible that (X^s, Y^t) holds due to one computation path and (Y^t, Z^r) due to another incompatible computation path, while (X^s, Z^r) does not

[1] Possibly because the analysis makes approximations and may overestimate the *ALIAS* relation. Also reflexive tuples exist, e.g. $(X^{(.,1)}, X^{(.1)})$ with X of type list expresses: if one element of the list is live then others are possibly live, in other words the value of X is a list with possibly several occurrences of the same element.

hold. When, due to a unification or a procedure call, new $ALIAS$ tuples are derived then an *alternating closure* which alternates over new and old tuples is used to update the $ALIAS$ relation. The $ALIAS$ relation consists of a global component GA containing the aliases between input arguments of the procedure that existed prior to calling the analysed procedure and a local componant LA_p giving the aliases created by executing the procedure up to the program point p. The whole $ALIAS$ relation is then given by the alternating closure of both components, i.e. $ALIAS_p = Altclos(GA, LA_p)$.

- A set IN_USE. It is the union of two sets. The first, $LIVE_0$, is a global component which contains elements X^s expressing that X^s is live due to the context of the caller. In other words, X^s will probably be accessed after completion (with success or failure) of the current procedure. It is called $LIVE_0$ because it is derived from the liveness which exists in the program point where the analysed procedure is called. The second set, LU_p, is a local component containing elements X^ϵ for every variable which can be accessed after completing the execution of the current atom (with success or failure).

The set $LIVE_p$ describing the liveness in a program point is obtained by using the implications in $ALIAS_p$ to extend the initial liveness set given by IN_USE_p. More formally, defining

$$\begin{aligned} \mathcal{L}(L, A) = & \ \{X^s | X^s \in L\} \\ & \cup \{X^s | Y^t \in L \text{ and } (Y^{t.t_1}, X^s) \in A\} \\ & \cup \{X^{t.s_1} | Y^{s.s_1} \in L \text{ and } (Y^s, X^t) \in A\} \end{aligned} \tag{1}$$

we have

$$LIVE_p = \mathcal{L}(IN\_USE_p, ALIAS_p) \tag{2}$$

where $IN\_USE_p = LIVE_0 \cup LU_p$ and $ALIAS_p = Altclos(GA, LA_p)$. For more technical details we refer to [3].

2.5 Reuse analysis

The liveness analysis computes a safe approximation of the liveness, hence a data structure which is not live is –if reachable at all– only reachable from the program variables in the current atom. The interesting case is when the current atom is a deconstruction $X \Rightarrow f(X_1, \ldots, X_n)$ and X^ϵ is not live in the current program point. Then, [3] the data structure X^ϵ is *available for reuse* in the program point following the deconstruction as it is no more reachable from any accessible program variable. If, within the procedure, the deconstruction is followed by a construction $Y \Leftarrow f(Y_1, \ldots, Y_n)$ then the top level of X can be used to construct Y and we say that there is *direct* reuse (of X^ϵ) in the analysed procedure. We say that a procedure has *indirect* reuse if it calls a procedure q/m and q/m allows for (in the context of that call) direct or indirect reuse. To distinguish this phase from the liveness analysis, we call it the *reuse analysis*. When analysing the program as a single unit, this pass requires no fixpoint iteration as it only has to verify whether deconstruction/construction pairs are fitted for reuse.

2.6 Using the analysis results

Which deconstruction/construction pairs are suited for reuse can be returned to the compiler as an extra piece of information in the HLDS code. The work of Taylor [18] could be adapted to perform the actual reuse. Actually this should be easier than the current approach which still requires a non-trivial local analysis of the programmer-provided *di/uo* annotations.

An important point to recall is that procedures can be called with different call patterns and that one obtains a form of multiple specialisation as described in [16]. Eventually, although having different call patterns, two versions may end up with identical direct and indirect reuse, hence should be merged in a single version. Such version control is discussed in [16]. As also explained there, the analysis has two phases. In our case, the first phase performs the liveness analysis and constructs a table with call/exit patterns (a pattern consists of the $ALIAS$ relation and the IN_USE set). This phase requires fixpoint iteration to handle recursive procedures. The second phase performs a single pass over the program, generates the required versions for each procedure and checks whether deconstruction/construction pairs satisfy the reuse condition. It also takes care that each procedure call calls the proper version.

3 Module based analysis

For separate compilation of modules Mercury uses interfaces which are created during compilation and which contain the needed information of the exported predicates. Mutual dependencies are not a problem as all relevant information about exported procedures is in the obligatory declarations. However, mutual dependencies pose a problem during abstract interpretation of mutually recursive predicates defined in different modules. This problem is discussed in Sect.3.3. For the time being, we assume a tree structure for the calling hierarchy between modules.

3.1 Modular liveness analysis

The domain of the liveness analysis is suited for a so called goal independent analysis [4]. Starting the analysis of a procedure p/n with an empty $LIVE_0$ and an empty GA component, one obtains as exit pattern the $ALIAS$ relation describing the possible aliases created between the arguments of p/n due to the execution of p/n. Similarly, one obtains the IN_USE component describing the arguments of p/n in use due to the execution of p/n (which are nothing else than the data structures in the arguments which could be needed when backtracking returns control to p/n). This goal independent information about p/n suffices to perform the liveness analysis of a procedure q/m calling p/n. Combining the goal independent exit pattern of p/n with the liveness information of the point before the call to p/n, one can compute the liveness in the program point following p/n[2]. Storing the call/exit patterns of exported predicates in the module interface, modules importing predicates can be analysed by accessing their call/exit pattern in the corresponding interfaces. Hence a goal independent liveness analysis can be performed in a modular way.

3.2 Modular reuse analysis

For a truly modular analysis, the reuse analysis must also be modular. Problem is that reuse is not goal independent. One useful property is that the amount of reuse decreases monotonically with increases of $LIVE_0$, the liveness in the caller's context and GA, the aliases in the caller's context. Hence performing a reuse analysis based on a goal independent liveness analysis of a procedure exported from a module gives a maximal amount of reuse. However, it is unlikely that an actual call will ever have $LIVE_0 = \emptyset$. At least the output arguments of the procedure will be in $LIVE_0$[3]. This suggests the following approach for analysing procedures exported from a module: besides the standard Mercury code without reuse, a version with reuse is generated. It is created starting from a liveness analysis of these procedures which are initialised with $GA = \emptyset$ and $LIVE_0 = \{X^\epsilon | X \textit{ is an output argument}\}$ (called the *default liveness analysis*). The standard reuse analysis is applied on the outcome of this analysis and results in code with reuse annotations for the exported procedure and for the internal procedures called by them. When calling an imported procedure, one can check whether the caller's context meets the assumption for reuse, i.e. whether there are no aliases between the (input) arguments of the procedure and none of the input arguments is live. If so, the version with reuse can be called; if not, the version without reuse is called. In what follows, we call the default liveness analysis of the procedures exported by a module followed by the reuse analysis the *default analysis* of the module.

The above approach is overly pessimistic. Consider predicate *append* with mode `(in,in,out)`. The default analysis reveals that the backbone of the first input list can be reused for constructing the output list. Calls with the elements of the first input list live, or with the second input list live do not meet the requirement for using the reuse version. However, performing the analysis for such call patterns reveals that reuse is still possible. Intuitively, the reuse version of *append* can be called when the backbone of the first input list is not live. More generally, the problem to be addressed is: which results about the default analysis should be stored and how to check that a particular call meets the requirements to call the reuse version resulting from the default analysis.

---

[2] Although the $ALIAS$ component is not idempotent, $Altclos(ALIAS, ALIAS) \neq ALIAS$, there is no loss of precision with respect to a goal dependent analysis. This is because the aliases created during the analysis of p/n depend only on the mode declared for p/n, not on the call pattern of p/n.

[3] Making a call and not using some of the outputs is not impossible; however, we consider it part of a preceding source level specialisation step to create a special version of the procedure with the unneeded outputs eliminated.

3.2.1 Conditions for reuse In what follows, a component is given a subscript i when its value depends on the program point i. It is given a superscript da or gd when its value differs between the default and the goal dependent analysis.

Consider the following program fragment:

```
q(Q1,...,Qn) :- ... i X => f(...), ...
```

Performing a default liveness analysis for q/n, formula (2) applied at program point i yields the live set

$$LIVE_i^{da} = \mathcal{L}(LU_i \cup LIVE_0^{da}, LA_i) \tag{3}$$

where $LIVE_0^{da} = \{Q_i^\epsilon | Q_i \text{ is an output argument of } q\}$ and LU_i, LA_i are respectively the local use and the local aliases at i. The top level of X becomes available for reuse when $X^\epsilon \notin LIVE_i^{da}$. A goal-dependent analysis of q/n computes

$$LIVE_i^{gd} = \mathcal{L}(LU_i \cup LIVE_0^{gd}, \mathit{Altclos}(LA_i, GA^{gd})) \tag{4}$$

with $LIVE_0^{gd}$ the initial liveness of q/n and GA^{gd} the initial aliases of q/n. We want to check whether reuse is still possible without performing a goal dependent analysis for q/n. To do so, we first present a theorem which addresses a slightly more general situation. First we introduce some new notation.

- With R a relation over (a set from) a domain of data structures and V a set of variables, $R|_V$ denotes the restriction of the relation to the variables in V.
- With D a set of data structures, V a set of variables and LA_i the local aliases at a program point i, D_i^V is the abbreviation for $\mathcal{L}(D, LA_i)|_V$, i.e. the restriction to V of the data structures reachable from the set D due to the aliases in LA_i.

Theorem 1. *Given a procedure with head variables $\mathcal{H}$ and local aliases LA_i and local use LU_i at program point i. Let D be a set of data structures from the procedure. If*

$$X^s \in D \text{ implies } X^s \notin \mathcal{L}(LU_i \cup LIVE_0^{da}, LA_i) \tag{5}$$

$$Y^t \in D_i^{\mathcal{H}} \text{ implies } Y^t \notin \mathcal{L}(LIVE_0^{gd} \cup LU_i^{\mathcal{H}}, \mathit{Altclos}(GA^{gd}, LA_i|_{\mathcal{H}})) \tag{6}$$

then

$$X^s \in D \text{ implies } X^s \notin \mathcal{L}(LU_i \cup LIVE_0^{gd}, \mathit{Altclos}(GA^{gd}, LA_i)) \tag{7}$$

Proof. Note that GA^{gd} and $LIVE_0^{gd}$ contain only data structures from $\mathcal{H}$. Assume $X^s \in \mathcal{L}(LU_i \cup LIVE_0^{gd}, \mathit{Altclos}(GA^{gd}, LA_i))$. $X^s \in LU_i \cup LIVE_0^{gd}$ is in contradiction with conditions (5) ($X^s \notin LU_i$) and (6) (if $X^s \in LIVE_0^{gd}$ then $X \in \mathcal{H}$ but then $X^s \in D_i^{\mathcal{H}}$) of the theorem. Hence there must be a path starting in $Z^t \in LU_i \cup LIVE_0^{gd}$, alternating over GA^{gd} and LA_i and ending in X^s (according to (1)). We perform a case analysis and show that each case is inconsistent with the conditions of the theorem.

– $Z \notin \mathcal{H}$ (hence $Z^t \in LU_i$)
  - path of length 1: $(Z^t, X^s) \in LA_i$ and thus (5) is violated for X^s.
  - path of length > 1: The first edge in the path is of the form $(Z^t, Y_1^{s_1})$ and belongs to LA_i; moreover (the path alternates with GA^{gd}) $Y_1 \in \mathcal{H}$ hence $Y_1^{s_1} \in LU_i^{\mathcal{H}}$. From the assumption it follows that $(Y_1^{s_1}, X^s) \in \mathit{Altclos}(GA^{gd}, LA_i)$, hence an alternating path $(Y_1^{s_1}, Y_2^{s_2}), \ldots, (Y_n^{s_n}, X^s)$ with $Y_1, \ldots, Y_n \in \mathcal{H}$ exists. If $X^s \in \mathcal{H}$ then we have $(Y_1^{s_1}, X^s) \in \mathit{Altclos}(GA^{gd}, LA_i|_{\mathcal{H}})$ and (6) is violated for X^s. Otherwise $X^s \notin \mathcal{H}$, hence $Y_n^{s_n} \in D_i^{\mathcal{H}}$ and $(Y_1^{s_1}, Y_n^{s_n}) \in \mathit{Altclos}(GA^{gd}, LA_i|_{\mathcal{H}})$ hence (6) is violated for Y^{s_n}.

– $Z \in \mathcal{H}$ (hence $Z^t \in LU_i|_{\mathcal{H}} \cup LIVE_0^{gd}$)
  - path of length 1: Either $(Z^t, X^s) \in GA^{gd}$ or $(Z^t, X^s) \in LA_i$. In the former case, $(Z^t, X^s) \in \mathit{Altclos}(GA^{gd}, LA_i|_{\mathcal{H}})$ and (6) is violated for X^s. In the latter case, either $Z^t \in LU_i$ and (5) is violated for X^s or $Z^t \in LIVE_0^{gd}$, hence also $Z^t \in D_i^{\mathcal{H}}$ and condition (6) is violated for Z^t.
  - path of length > 1: $(Z^t, X^s) \in \mathit{Altclos}(GA^{gd}, LA_i)$ hence there is an alternating path $(Y_1^t, Y_2^{s_2}), \ldots, (Y_n^{s_n}, X^s)$ with $Z = Y_1$ and $Y_1, \ldots, Y_n \in \mathcal{H}$. If $X^s \in \mathcal{H}$ then $(Y_1^t, X^s) \in \mathit{Altclos}(GA^{gd}, LA_i|_{\mathcal{H}})$ and condition (6) is violated for X^s. Otherwise $X^s \notin \mathcal{H}$, hence $Y_n^{s_n} \in D_i^{\mathcal{H}}$ and $(Y_1^t, Y_n^{s_n}) \in \mathit{Altclos}(GA^{gd}, LA_i|_{\mathcal{H}})$ hence condition (6) is violated for $Y_n^{s_n}$.

$\square$

With $D = \{X^\epsilon\}$, the set of data structures potentially available for reuse, condition (5) verifies whether X^ϵ is available for reuse in the default analysis, while condition (7) verifies whether X^ϵ is available for reuse in the goal dependent analysis. Hence, given that reuse is possible in the default case, the theorem says that the latter condition can be replaced by condition (6) which is a condition involving only head variables. It suffices to store the relations/sets $LA_i|_{\mathcal{H}}$, $D_i^{\mathcal{H}}$ and $LU_i^{\mathcal{H}}$ (the *reuse information*) in the module interface to allow a caller to check whether it can call the reuse version of the imported procedure q/n.

Consider $\mathit{append}(A, B, C)$ with mode `(in,in,out)`, $\mathcal{H} = \{A, B, C\}$ and $LIVE_0^{da} = \{C^\epsilon\}$. The default analysis computes in the point preceding $A \Rightarrow \ldots$ that $D = \{A^\epsilon\} = D_i^{\mathcal{H}}$, $LA_i|_{\mathcal{H}} = \emptyset$ and $LU_i^{\mathcal{H}} = \{B^\epsilon, C^\epsilon\}$. Consider a call with $LIVE_0^{gd} = \{A^{(\cdot,1)}, C^\epsilon\}$ and $GA^{gd} = \emptyset$. Condition (6) becomes: $Y^t \in D_i^{\mathcal{H}}$ implies $Y^t \notin \mathcal{L}(\{A^{(\cdot,1)}, B^\epsilon, C^\epsilon\}, \emptyset)$ which is satisfied.

We also have to address the problem of verifying whether a version with indirect reuse can be called. Consider the following fragment:

```
r(R1,...,Rm) :- ... j q(X1,...,Xn), ...
q(Q1,...,Qn) :- ... i X => f(...), ...
```

To verify that the default version of r/m can call the version of q/n with reuse at program point i, one should check that the data structures in $D' = \rho(D_i^{\mathcal{H}_q})$ (ρ renames the variables in the head of q/n into the variables in the call to q/n) are not live, taking into account $LA'_j = LA_j \cup \rho(LA_i|_{\mathcal{H}})$ as local aliases and

$LU'_j = LU_j \cup \rho(LU_i^{\mathcal{H}})$ as local use (this is the rephrasing of (6) to the context of program point j). If this indirect reuse is possible in the default case, then the reuse information over the variables of r/m ($LA'_j|_{\mathcal{H}_r}$, ${D'_j}^{\mathcal{H}_r}$, and ${LU'_j}^{\mathcal{H}_r}$) can be computed. A caller to r/m can use the version of r/m with indirect reuse in q/m if (6) is satisfied for the reuse information of r/m. Repeating this, the reuse information can be propagated up to the level of the exported predicates. Note that here fixpoint iteration is needed to handle recursive procedures (the reuse information of a recursive procedure is needed while computing it).

3.2.2 Version control The default analysis of an exported procedure may reveal more than one point for reuse: there can be several deconstruction/construction pairs allowing reuse as well as indirect reuse through calls to procedures with reuse. For each case of reuse, a relation $LA_i|_{\mathcal{H}}$ and sets $D_i^{\mathcal{H}}$ and $LU_i^{\mathcal{H}}$ are obtained so that the condition for the particular reuse can be tested by a caller. (The reuse is unconditional when $D_i^{\mathcal{H}}$ is empty.) If all conditions are satisfied then the version of the exported procedure which performs all the reuses can be called. If one of the conditions is violated, should one give up all reuse and call the version without any reuse? Ideally one should call a version which performs all the other reuses. This would require to create 2^n versions of the procedure when there are n places with (different) conditional reuse. This becomes infeasible for large n. Experience will have to show whether many cases of conditional reuse show up in a single procedure. If so, some pragmatic solutions will have to be worked out. Several scenarios are feasible:

- Only make 2 versions, one without any conditional reuse and one with all conditional reuses.
- Provide the programmer with a pragma to express interest in the reuse of a data structure. Then only the versions with conditional reuse involving these data structures are created. Such a pragma should still be much less tedious to use than *di/uo* declarations.
- Make versions on demand: reuse code is only generated if a call occurs for which a particular set of reuse conditions is satisfied. This scenario breaks with the strict modularity of the compilation process. Also, versions created on demand for the compilation of another module may finally never be called from the main program.
- Create all possible versions and add a final *global* analysis pass: once all modules of a program are analysed, a final global compilation can be performed, which, starting from the top-level module, makes a single pass over the whole program (and thus compiled code), and links those versions of the procedures which are actually needed into the final compiled code. While this solution also departs from the strict modularity principle, once modules are compiled with all their multitude of procedure versions, these modules will not have to be recompiled if used for other programs.

3.3 Circular dependencies between modules

A problematic circular dependency exists when predicates that depend on each other are distributed over different modules. Several solutions with different computational costs are feasible (see also the discussion in [15]).

- When importing a procedure from a module which cannot be analysed due to a circular dependency, a worst case assumption can be made for the liveness (all arguments are live) and for the aliasing (all pairs of data structures with compatible types are aliases) of the imported procedure. This allows to continue the analysis of the current module but gives a suboptimal result.
- Another solution is to make a best case assumption (only the output arguments are live and there are no aliases) and to reanalyse the current module when this assumption turns out to be false. So a fixpoint iteration across modules will be necessary.
- A third solution is to load all dependent modules and analyse them together as one module. If the module is too large, the call graph of its procedures could be used to split it into parts that can be compiled separately.

4 Experimental results

A prototype of the modular analysis has been implemented. The first phase performs a default liveness analysis of the exported procedures and the second phase uses its results to perform a reuse analysis. It creates for each exported procedure two versions (unless both versions are identical): one version with unconditional reuse only (or none at all) and a version with all possible reuse. For the latter version, the reuse information is also computed. It allows callers to verify whether they meet the conditions for reuse.

4.1 Benchmarks and Results

The following modules are analysed: basic library modules for tree and list manipulation (assoc_list, bintree, bool, bt_array, list, set_ordlist, tree234), library modules which import procedures from the basic ones (bag, bintree_set, eqvclass, graph, group, map, multi_map, queue, set, set_unordlist), a module of the industrial users of Mercury in the ESPRIT project ARGo (argo_cnters) and modules from the Mercury compiler (labelopt, llds, opt_util). In table 1 the library modules are in the upper part and the other modules in the lower part.

Time The time in seconds of the default liveness analysis[4].
Pr The number of procedures in the module.
Xp The number of exported procedures.
CR The number of exported procedures for which conditional reuse is detected.

[4] The analysis is done with a generic abstract interpretation tool written in Prolog, using Master Prolog, release 4.1 ERP on a UltraSPARC-IIi (333MHz) with 256MB RAM, using SunOS Release 5.7, under a usual workload.

module	Time	Pr	Xp	CR	UR	NR	Cnd	pol	DC	%DR	Lc	%LR	Ec	EcR	%ER
assoc_list	0.27	7	6	5	0	1	2.40	1.00	7	100	7	86	10	0	-
bag	2.46	26	24	17	10	5	4.95	1.19	4	100	29	90	70	25	100
bintree	1.37	30	19	6	2	11	3.00	1.03	17	100	56	43	33	0	-
bintree_set	0.16	23	21	11	6	9	1.08	1.04	1	100	16	75	26	6	100
bool	0.01	5	5	2	0	3	2.00	1.00	2	100	2	100	0	0	-
bt_array	5.97	37	17	6	4	9	1.75	2.30	38	97	137	55	181	0	-
eqvclass	1.55	21	12	4	4	4	1.00	1.19	5	100	27	78	37	12	100
graph	6.96	26	14	6	6	6	0.75	1.38	9	100	42	33	51	12	100
group	1.95	20	11	2	4	7	0.50	1.25	4	100	27	41	27	8	100
list	1.84	66	56	34	3	22	1.76	1.21	46	91	111	64	38	2	100
map	6.69	42	38	16	4	18	3.65	1.24	2	100	39	54	56	16	100
multi_map	1.95	36	33	11	6	16	2.82	1.03	1	0	13	46	48	21	100
queue	0.18	12	11	5	1	6	2.80	1.00	4	100	2	100	12	8	100
set	0.08	27	27	6	0	21	1.00	1.00	0	-	0	-	27	6	100
set_ordlist	0.52	31	29	6	0	23	1.00	1.16	7	0	41	2	19	6	100
set_unordlist	0.34	30	27	13	3	14	1.15	1.10	1	100	25	64	12	6	100
tree234	1140	74	20	10	1	9	5.55	1.14	283	88	443	71	136	0	-
argo_cnters	4.79	18	1	0	1	0	0.00	1.00	32	100	41	29	56	0	-
labelopt	12.00	6	2	2	1	0	2.00	2.00	2	100	17	88	13	9	78
llds	0.08	7	7	0	0	7	-	1.00	0	-	6	0	2	0	-
opt_util	2028	73	46	20	8	23	2.04	1.47	48	83	284	28	70	24	100

Table 1. Results; "-" means not applicable.

UR The number of exported procedures for which unconditional reuse is detected. (The two types of reuse can occur in the same procedure.)

NR The number of exported procedures without reuse.

Cnd The average number of reuse conditions for exported procedures with conditional reuse. Some conditions might be equal (i.e. different sources of reuse might produce exactly the same reuse information). The number of reduced conditions is not available here.

pol The average polyvariance resulting from the liveness analysis, i.e. the average number of call patterns per procedure.

DC The number of deconstruction/construction pairs in the analysed module.

%DR The percentage of deconstruction/construction pairs resulting in direct reuse.

Lc The number of calls to procedures internal at the module.

%LR The percentage of Lc that calls a version with reuse.

Ec The number of calls to imported procedures.

EcR The number of calls to imported procedures for which a reuse version exists.

%ER The percentage of EcR that calls a version with reuse. ("-" if EcR=0).

4.2 Discussion

– In most cases the analysis time is of the same order as the compilation time of the module. In a few cases it is rather large (tree234 and opt_util). We do not consider these times as unbearable, especially for library modules which have to be compiled only once. Modules importing them need only to consult the module interface. Our prototype leaves room for much improvement and it is a topic of further research to consider alternative representations of the *ALIAS* information and to develop appropriate widenings. Our reuse analysis is not yet fine-tuned enough to report reuse analysis times. In most cases the time needed for it is comparable to the time for the default liveness. We believe it should only be a small fraction of the total analysis time, especially in cases where the default analysis time is large.

- Reuse versions are created for a large fraction of exported predicates. Even unconditional reuse is quite frequent. This is an indication that our analysis is able to find an interesting amount of reuse.
- The average number of reuse conditions is in general small. However for bag and tree234 the average is about 5 conditions. In tree234 there is a procedure with 12 conditions. Yet some of these conditions appear to be equivalent, and upon simplification, the total set of different conditions is reduced to only 3. It seems not feasible to create a version for each of the combinations of reuse. It requires further experimentation to find out how many of the conditions are satisfied by typical calls to procedures with many conditions and what are good strategies for version creation (see discussion in 3.2.2).
- The polyvariance is in general low. The multiple specialisation inherent at the analysis does not result in an explosion of versions of the same procedure.
- Quite a large fraction of deconstruction/construction pairs result in direct reuse. If no direct reuse is detected, it is either because no reuse is possible or because the analysis is too imprecise. It requires hand analysis to find out what is the real cause; it is not feasible to analyse this on a large scale.
- Versions of local procedures with reuse are quite frequently called.
- Although the total number of calls to imported procedures (Ec) looks high with regard to the number of calls to such procedures for which a reuse version exists (EcR), most of the calls happen to be I/O related, or integer-operations, hence are to procedures which cannot have reuse versions.

4.3 Effect on the performance of Mercury programs

There is yet no version of the Mercury compiler which makes use of the reuse analysis. However, Taylor [18] has implemented structure reuse for a separate distribution of the Mercury compiler. It is based on the *di/uo* annotations provided by the programmer. Hand translating the reuse annotations in *di/uo* annotations, we did a few experiments.

For *argo_cnters*, a benchmark counting various properties in a file, this comparison revealed a speedup of up to 30% depending on the size of the input. Comparing memory usage yields an improvement of 50% (7 mega-words compared to 14 mega-words in the non-optimized version). Other programs (too small to be included in table 1), like *nrev* (naive reverse of a large list of integers), and *insert* (inserting elements in a binary tree) reveal speedups up to 75%, and memory savings up to 85%. The latter programs were also used as benchmarks in [18]. The *di/uo* annotations considered there correspond to the reuse annotations we derived.

5 Conclusion

This paper describes how the liveness analysis of [11] can be extended into a modular reuse analysis for Mercury. The reuse analysis discovers when data structures become garbage and allows the compiler to reuse them. The concepts can be adapted for other sequential logic programming languages such as Ciao Prolog and HAL; however, one can expect that less precise type, mode and determinism information will result in less reuse.

The contribution of this paper is to develop a modular reuse analysis which can become part of module compilation. While this was fairly straightforward for the liveness analysis —liveness analysis is fitted for goal independent analysis—, it was a nontrivial task for reuse analysis. A major contribution is a theorem which allows to transform a condition verifying whether data structures become available for reuse at some program point into a condition over the head variables of the procedure which can be verified by the caller of the procedure. This is the basis to derive conditional reuse for procedures exported by a module. Storing the necessary information in the module interface, other modules importing the procedure can easily verify whether they can call a version with reuse or not.

The paper also reports on experiments done with a prototype. Our results are promising; a substantial number of opportunities for reuse are detected and a few experiments show that they can have a substantial impact on performance and memory consumption. The results are encouraging enough to start the development of an analyser in Mercury itself which can become a component of the Mercury compiler. Much more extensive experiments will localise the remaining problems. We expect that we will be confronted with too bulky aliasing information and will have to consider more compact representation for it and/or appropriate widening operators. The issue of multiple specialisiation will also need further investigation. We also have to incorporate language features ignored so far such as higher order predicates and type classes.

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. Maurice Bruynooghe. A practical framework for the abstract interpretation of logic programs. *Journal of Logic Programming*, 10(2):91–124, February 1991.
3. 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.
4. M. Codish, M. Bruynooghe, M. García de la Banda, and M. Hermenegildo. Exploiting goal independence in the analysis of logic programs. *Journal of Logic Programming*, 32(3):247–261, September 1997.
5. 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.

6. B. Demoen, M. García de la Banda, W. Harvey, K. Marriott, and P. Stuckey. An overview of HAL. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, pages 174–188, Virginia, USA, October 1999. Springer Verlag.
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. M. V. Hermenegildo, F. Bueno, G. Puebla, and P. López. Program analysis, debugging, and optimisation using the Ciao preprocessor. In D. De Schreye, editor, *Logic programming, Proc. of the 1999 Int. Conf. on Logic Programming*, pages 52–66, Las Cruces, NM, December 1999. MIT-Press.
10. 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.
11. 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.
12. 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.
13. N. Mazur, G. Janssens, and M. Bruynooghe. Towards memory reuse for Mercury. In *Proc. Int. Workshop on Implementation of Declarative Languages (IDL'99)*, Paris, September 1999.
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. G. Puebla and H. Hermenegildo. Some issues in analysis and specialisation of modular programs. In M. Leuschel, editor, *Optimisation and Implementation of Declarative Programs, (WOID'99), a ICLP'99 workshop*, 1999. 17 pages.
16. G. Puebla and M. Hermenegildo. Abstract multiple specialisation and its application to program specialisation. *The Journal of Logic Programming*, 41:279–316, November-December 1999.
17. 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.
18. Simon Taylor. Optimization of Mercury programs. Honours report, Department of Computer Science, University of Melbourne, November 1998.
19. Mads Tofte and Talpin Jean-Pierre. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.
20. K. Ueda. Linearity analysis of concurrent logic programs. In M. Leuschel, editor, *Optimisation and Implementation of Declarative Programs, (WOID'99), a ICLP'99 workshop*, 1999. 14 pages.
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.