

Towards memory reuse for Mercury

*Nancy Mazur
Gerda Janssens
Maurice Bruynooghe*

Report CW 278, June 1999



Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Towards memory reuse for Mercury

*Nancy Mazur
Gerda Janssens
Maurice Bruynooghe*

Report CW 278, June 1999

Department of Computer Science, K.U.Leuven

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 has not to worry about the details of memory management.

The paper reports on some experiments with a prototype of an 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. The prototype has been applied on a module of the Mercury compiler and finds most of the opportunities for reuse. With some modifications, it will be able to find all.

The paper also develops the principles of a module based analysis which is essential for the analysis of large Mercury programs with code distributed over many modules.

Keywords : Program Analysis, Mercury, Liveness, Compile-time Garbage Collection

CR Subject Classification : D.3.4, I.2.3

Towards memory reuse for Mercury

Nancy Mazur, Gerda Janssens, 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 has not to worry about the details of memory management.

The paper reports on some experiments with a prototype of an 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. The prototype has been applied on a module of the Mercury compiler and finds most of the opportunities for reuse. With some modifications, it will be able to find all.

The paper also develops the principles of a module based analysis which is 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 [16] 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 has not 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, in practice 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, 13, 14], as well as functional programming languages [2, 11, 17, 18]. Some of the approaches involve special language constructs (such as uniqueness declarations within Mercury) [1, 16, 21, 22], others are based on compiler analyses [7, 12]. Mulkers et al. [15] have developed such an analysis for Prolog, however, the lack of declarations and the impurity of Prolog offer little perspective for integrating 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 reports on a prototype implementation of a live-structure analysis for Mercury. It describes the outcome of an experiment where a module from the Mercury compiler has been analysed. To achieve the long term goal of integrating the analysis in the Mercury compiler, a module based analysis is necessary. The paper develops the concept 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 the experiment with the analysis of a module of the compiler and describes the work needed to improve the analysis. In section 4 the module based analysis is developed. We conclude with a brief discussion in Section 5.

2 Preliminaries

Mercury is a logic programming language provided with types, modes and determinism declarations. The language is strongly typed and its type system is based on a polymorphic many-sorted logic [8]. Types are of particular importance to us. A type t (or if polymorphic $t(T_1, \dots, T_n)$ with T_1, \dots, 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. Such a tree has two kinds of nodes: type nodes and constructor nodes. A type node is labelled with either a type variable or a type name. If labelled with a variable, it has no children; if labelled with a type name, it has constructor nodes as children, one for each of the type constructors in the definition of the type (the order is unimportant). Constructor nodes are labelled by the constructor name and have an (ordered) set of type nodes as children: one for each argument of the constructor in the declaration. While type trees of recursive types are infinite, we consider equivalence classes over the

typenodes: two nodes on a path from the root are equivalent when they have the same label. In this way, any type tree has a finite representation as a type graph. Nodes have an associated type, namely the type of the tree with the node as root.

Example 2.1 *The polymorphic type $\text{list}(T)$ is defined as:*

$\text{list}(T) \text{ ---> } [] ; [T|\text{list}(T)].$

Its type graph is shown in Fig. 1.

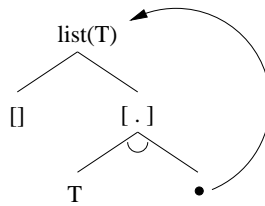


Figure 1: Type graph of $\text{list}(T)$

Type selectors are used to select a node in a type tree (and through the type associated with the node, a subtype of a type). ϵ denotes the empty selector and t^ϵ selects the root node of t . With $c(t_1, \dots, t_n)$ one of the alternatives in the type definition of t , $t^{(c,i)}$ ($i \leq n$) selects the i^{th} child of the child of t with c as label (and the type associated with it). As concatenator of type selectors we use “.”. With s a selector applicable on t_i , $t^{(c,i).s}$ selects t_i^s . We also define equivalence classes over type selectors. In the context of a type t we have that $s_1 \equiv s_2$ when t^{s_1} and t^{s_2} select the same node. Also the number of these equivalence classes is finite. For example, in the context of type $\text{list}(T)$, and using “.” as list constructor, $\epsilon \equiv (.,2)$ and $\text{list}(T)^\epsilon = \text{list}(T)^{(.2)}$.

Liveness of a variable is expressed at the level of the type nodes of its type graph. With t the type of X , $\text{Live}(X^s)$ expresses that the subterm of X corresponding to the subtype t^s is live. Parts of a variable which are not live are available for reuse. For example, with X of type $\text{list}(T)$, three cases can be distinguished: (1) $\text{Live}(X^\epsilon)$ expressing that the whole value of X is live; (2) $\text{Live}(X^{(.1)})$ expressing that only the elements of the list are live (and the backbone of the list is available for reuse); (3) nothing is live, i.e. everything is available for reuse. Formally, liveness can be defined as a total mapping from the type nodes to $\{\text{true}, \text{false}\}$ with the constraint that $\text{Live}(X^s)$ implies $\text{Live}(X^{s.t})$ for all selectors s and t . These derived live nodes are left implicit (as in the case (1) above where also $\text{Live}(X^{(.1)})$ holds).

Our analysis is performed at the level of the High Level Data Structure (HLDS) constructed by the Mercury compiler. It is a kind of normal form

which has one definition for each mode of each program predicate. The bodies of the clauses in this normal form contain conjunctions, disjunctions and if-then-else's. Unifications are made explicit, all atoms about program predicates have distinct variables as arguments and special atoms are used for unification. Four cases of unification are used. They are: (1) test $X == Y$, (2) assignment $X := Y$, (3) construction $X \Leftarrow f(Y_1, \dots, Y_n)$, and (4) deconstruction $X \Rightarrow f(Y_1, \dots, Y_n)$ [8]. In what follows, we use the term head variables for the arguments (variables) which are in the head of the normal form.

Our analysis 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 operation on concrete data by abstract operations over data descriptions. The analysis of a procedure call consists of procedure-entry, the execution of the statements in the procedure's body and procedure-exit. The analysis computes an abstract state in each program point and uses fix-point iteration to cope with recursion. It is a polyvariant analysis: it analyses predicate definitions for each call pattern which shows up during the analysis.

In the live-structure analysis of [4], the liveness information in a program point (preceding the *current* atom) is derived from three components associated with this program point:

- Forward use (*FU*): which variables can be accessed by the forward execution of the program *after* successful completion of the current atom (a set of variables).
- Backward use (*BU*): which variables can be accessed upon backtracking assuming the current atom fails (a set of variables).
- Aliases (*Alias*): which sharing is possible between the data structures representing the values of the bound variables in the program point (before executing the current atom) (a set of pairs (X^{sx}, Y^{sy})).

Each of these three components is separated in a global component and a local component. The global component describes the information coming from the context of the caller. Assuming q is called from r with actual arguments X_1, \dots, X_n and assuming formal arguments Y_1, \dots, Y_n , the global components describe the head variables Y_i corresponding to the actual arguments X_i which are in forward/backward use in the program point before the call (*GFU* and *GBU*) and the aliases between head variables Y_i and Y_j which correspond to the aliases between X_i and X_j existing in that program point (*Galias*). The local components describe the extra information gathered during the analysis of the body of q (*LFU*, *LBU*, and *Lalias*).

As explained in [4], these three components are used to compute the liveness in a program point. This liveness does not correspond exactly to what is explained above but is rather a "strong" liveness as it ignores the

accesses to data structures made by the instruction following the program point. So if X^s is not live in program point p , it means there are no access paths to X^s apart from those in the next instruction. The purpose of the analysis is to find opportunities for reuse; more concrete, the interest is in deconstruction statements $X \Rightarrow \dots$. If X^ϵ is not (strongly) live in the preceding program point, then the top level data structure (e.g. the first listcell when X is of type list) is available for reuse. The most convenient way to indicate this to the compiler is to insert a pragma *reuse*(X) in the HLDS at the program point following the deconstruction. In general it is also possible that X^ϵ is live in the point before the deconstruction but no longer in a later program point, in which case the pragma can be inserted at that point. Finally, reporting the possibility for reuse only makes sense if the clause contains a construction statement which can really reuse the cell available for reuse. This is a matter of inspecting the remainder of the clause (or also some preceding code if some code movement is possible). This aspect was handled manually in the experiments described in the paper.

The initial liveness upon entry of a clause definition is denoted $Live_0$. It is computed by procedure-entry which projects the liveness in the program point before the call on the arguments of the call and applies the renaming between formal and actual parameters.

3 A case study: *labelopt*

3.1 Analysis engine

For our experiments we are using the AMAI [10] as engine for abstract interpretation. The original engine, written in Prolog, has been extended with the necessary functionality to handle disjunctions (if-then-else constructs are transformed into deterministic disjunctions). The abstract domain is the domain for live-structure analysis as sketched in Section 2. The engine is an abstract machine which executes the code of the abstracted program. To generate the code of the abstracted programs, the Mercury compiler (v0.8) has been extended with a module which converts the intermediate HLDS structures produced by the compiler into AMAI instructions. These instructions also contain the necessary information about types, modes and determinism which is needed for the live structure analysis. The engine is able to perform a goal dependent analysis of a call to a predicate, given that the code includes all predicate-definitions on which the called predicate depends (directly or indirectly). The analysis is polyvariant and the result consists of call-exit pairs for all predicates needed for the analysis of the top level predicate call and, for each call pattern, an annotated version giving the abstract state in each program point. From this information it is pretty straightforward though tedious, to derive the *reuse* pragma's by hand (soon to be implemented).

3.2 The analysis of *labelopt*

The analysed program, *labelopt*, is a module from the Mercury compiler. The main predicate exported by this module is `labelopt_main`.

The purpose of this predicate is to transform a list of program instructions into a new list of optimised instructions. The module calls predicates from two other modules: *opt_util* and indirectly *list*. As the output data produced by the analysis engine is quite cumbersome to review (manually), we limited our analysis by substituting the few predicates from *opt_util* by dummy predicates, such that the liveness results for the main predicates within *labelopt* were not influenced. We kept the code for the list manipulation predicates though, as these presented a high potential for reuse.

3.2.1 Call graph and potential reuses

Figures 2 and 3 show the (simplified) call graph of `labelopt_main`, leaving out dummy calls. Predicates whose definitions contain pairs of deconstruct/construct instructions (candidates for reuse) are marked with *D/C*. Recursive calls are indicated by loops in the call graph. A brief description follows.

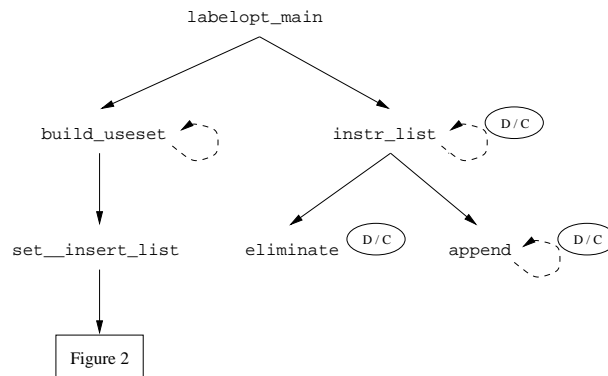


Figure 2: Call graph of `labelopt_main`.

labelopt_main: Input is a list of instructions; a new list is the output. There is no deconstruct/construct pair within the body. Reuse will have to occur at a lower level, if at all.

build_useset: Input is a list of instructions and a set of labels (initially empty); a new set of labels is the output. Again, there is no deconstruct/construct pair present within the body. Reuse at this level is also not possible.

instr_list: The input, a list of instructions, is explicitly deconstructed. For each instruction a new list of instructions is generated, either ex-

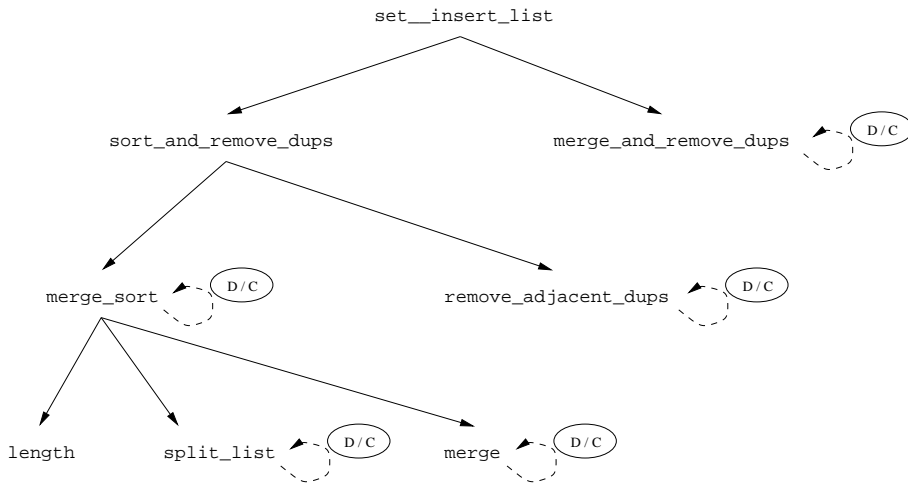


Figure 3: Call graph of `set_insert_list/3`. All predicates belong to the list-module, with exception of `set_insert_list`.

explicitly or by calling `eliminate`. Each of these lists is then prepended to the list of instructions computed by the recursive call. In this body we have explicit deconstruction and construction. It is a candidate for reuse.

eliminate: Basically a variable of type pair is deconstructed and a new pair is created. The output of this predicate is a list containing this new pair. There is an explicit deconstruct/construct, so it is a candidate for reuse.

list manipulation predicates: The remaining predicates are predicates which essentially manipulate lists. The meaning of each of these predicates should be quite obvious. Some of them have explicit deconstruction/construct pairs, and are therefore candidates for reuse.

Judging solely on the presence of explicit deconstructions/constructions, the following predicates are candidates for reuse: `instr_list`, `eliminate`, `append`, `merge_sort`, `split_list`, `merge`, `remove_adjacent_dups`, `merge_and_remove_dups`. Figures 4 and 5 present the relevant pieces of the definitions of these predicates as given by the HLDS code of the compiler.

3.2.2 Identified reuses

Our analysis detects reuse of the top level of the input structure in the predicates `append`, `instr_list`, `remove_adjacent_dups`, `eliminate` and `split_list`. Within each of these predicates a variable L is deconstructed ($L \Rightarrow f(\dots)$) where L^ϵ is not live in the program point before that deconstruction. For all predicates where lists are involved with exception of `split_list`

```

:- pred instr_list(list(instruction),
                  list(instruction)).
:- mode instr_list(in,out) is det.
instr_list([],[]).
instr_list(L1,L2):-
  L1 => [ IO | MoreL ],
  IO => Uinstr - _Comment,
  ( ... % perform some tests
  -> R <= [ IO ]
  ; eliminate(IO,R),
    opt_util_predicate(Uinstr,...)
  ),
  instr_list(MoreL,MoreR),
  append(R,MoreR,L2).

:- pred eliminate(instruction,
                  list(instruction)).
:- mode eliminate(in,out) is det.
eliminate(I,List) :-
  I => Instr0 - Comment0,
  ( ... % perform some tests
  -> List <= []
  ; NewInstr = ... , % some constant
    NewI <= NewInstr - Comment0,
    List <= [NewI]
  ).

```

Figure 4: Relevant code of *labelopt*-predicates.

the complete input list becomes consumed through the recursive calls. In `split_list`, the base case can be reached before the complete list is consumed, in which case the second output argument shares directly the whole input list. In `merge_sort`, we have a slightly different situation. The input list $L1$ is still live in the program point before the deconstruction; however, if we check the program point before the construction $L2 \leq [E]$, we see that $L1^\epsilon$ is neither live nor an alias of $L2$ or E , so the top level of $L1$ is available for reuse at that point.

3.2.3 Missed reuses and the remedy

In `merge` and the very similar `merge_and_remove_dups`, the reuse is missed by our analysis. We restrict the discussion to `merge`. The call pattern of the call `merge(A,B,C)` contains no aliases and only the output argument C is live. Inspection of the code shows that either the top level of A or the top level of B can safely be reused for constructing a cell of the output C . The problem is that A (and B) are live in the program point preceding their deconstruction. The analysis identifies A^ϵ with $A^{(\cdot,2)}$. As a consequence, A and its tail which is stored in Xs become aliases. Xs is live in every program point of the disjunct with the recursive call `merge(Xs,B,Zs)` and because of the imprecise alias, so is A . In the other disjunct, the similar observation holds for the input argument B . A solution is to replace the first deconstruct by $A \Rightarrow [X|_]$ and to insert a second deconstruct $A \Rightarrow [_|Xs]$ in front of the instruction using Xs (in this case in front of `merge(Xs,B,Zs)`). With this code, our analysis detects that A is available for reuse in the program point preceding the second deconstruct. Another solution to this problem is not to identify X^ϵ with $X^{(\cdot,2)}$ for lists (and similarly for other recursive data structures), at least not for variables which are used in a deconstruction operation. This solution requires a revision of our abstract domain and will result in slightly larger abstract states; however, if limited to variables which are candidates for reuse, the cost could be quite acceptable.

```

:- pred append(list(T),list(T),list(T)).
:- mode append(in,in,out) is det.
append(X,Y,Z):- X == [], Z := Y.
append(X,Y,Z):-
    X => [ X1 | Xs ],
    append(Xs,Y,Zs),
    Z <= [ X1 | Zs ].

:- pred merge_sort(list(T),list(T)).
:- mode merge_sort(in,out) is det.
merge_sort(L1,L2):- L1 == [], L2 <= [].
merge_sort(L1,L2):-
    L1 => [ E | R ],
    (
        % cannot fail switch on R
        % R = []
        L2 <= [ E ]
    );
    length(L1,Length),
    HL is Length // 2,
    (
        split_list(HL,L1,F,B),
        merge_sort(F,SF),
        merge_sort(B,SB),
        merge(SF,SB,L2)
    );
    % error
).

:- pred split_list(int,list(T),
                  list(T),list(T)).
:- mode split_list(in,in,out,out)
    is semidet.
split_list(N, List, Start, End) :-
    ( N == 0
    -> Start <= [],
        End := List
    );
    N1 is N - 1,
    List => [Head | List1],
    split_list(N1, List1, Start1, End),
    Start <= [Head | Start1]
).

:- pred merge(list(T),list(T),list(T)).
:- mode merge(in,in,out) is det.
list_merge(A, B, C) :-
    ( A => [X|Xs] ->
      ( B => [Y|Ys] ->
        ( compare(<, X, Y) ->
          Z := X,
          list_merge(Xs, B, Zs)
        );
        Z := Y,
        list_merge(A, Ys, Zs)
      ),
      C <= [Z|Zs]
    );
    C := A
).

:- pred remove_adjacent_dups(list(T),
                             T,list(T)).
:- mode remove_adjacent_dups(in,in,out)
    is det.
remove_adjacent_dups(L1,X,L2):-
    ( % cannot fail switch on L1
      L1 == [], L2 <= [X]
    );
    L1 => [ X1 | R ],
    ( X == X1
    -> remove_adjacent_dups(R,X,L2)
    ; remove_adjacent_dups(R,X1,MR),
      L2 <= [ X | MR ]
    )
).

:- pred merge_and_remove_dups(list(T),
                              list(T),list(T)).
:- mode merge_and_remove_dups(in,in,out)
    is det.
merge_and_remove_dups(A, B, C) :-
    % code very similar to merge/3
    merge(A,B,C).

```

Figure 5: Relevant code of *set* and *list* predicates.

3.3 Overview and cost

Table 1 presents the analysis results of the predicates as they appear in our experiment (see Appendix A for the code listing). Each of these predicates is annotated with reuse-information, call- and exit-pattern. The reuse-information is either r_d (direct reuse), r_i (indirect reuse), r_{d+i} (direct and indirect reuse), no (no reuse possible), no^* (missed reuse). The exit pattern consists of a life component and an alias component, both are relative to the life and alias components of the call pattern. Note that all predicates are deterministic (or semideterministic), therefore no variables will appear in backward use.

Our experiments were done on an UltraSPARC-IIi (333Mhz) with 256MB

Predicate	R.	Call patt. <i>Live0 - Alias0</i>	Exit patt. <i>Live+ - Alias+</i>
mylabelopt_main(H1,H2,H3,H4)	r_i	$\{H3, H4\} - \emptyset$	$\emptyset - \emptyset$
build_useset(H1,H2)	r_i	$\{H1, H2\} - \emptyset$	$\emptyset - \emptyset$
build_useset_2(H1,H2,H3)	r_i	$\{H1, H3\} - \emptyset$	$\emptyset - \emptyset$
instr_list(H1,H2,H3,H4,H5)	r_{d+i}	$\{H4, H5\} - \emptyset$	$\emptyset - \emptyset$
eliminate(H1,H2,H3,H4)	r_d	$\{H2, H3, H4\} - \emptyset$	$\{H1^{(-1)}\} - \{(H1^{(-1)}, H3^{(-1)})\}$
eliminate(H1,H2,H3,H4)	r_d	$\{H1^{(-1)}, H3, H4\} - \emptyset$	$\emptyset - \{(H1^{(-1)}, H3^{(-1)})\}$
set_init(H1)	no	$\{H1\} - \emptyset$	$\{H1\} - \emptyset$
set_insert_list(H1,H2,H3)	r_i	$\{H3\} - \emptyset$	$\{H1.H2^{(\cdot,1)}\} -$ $\{(H1, H3), (H2^{(\cdot,1)}, H3^{(\cdot,1)})\}$
set_member(H1,H2)	no	$\{H1, H2\} - \emptyset$	$\emptyset - \emptyset$
sort_and_remove_dups(H1,H2)	r_i	$\{H2\} - \emptyset$	$\{H2^{(\cdot,1)}\} - \{(H1^{(\cdot,1)}, H2^{(\cdot,1)})\}$
merge_and_remove_dups(H1,H2,H3)	no*	$\{H3\} - \emptyset$	$\{H1, H2\} - \{(H1, H3), (H2, H3)\}$
member(H1,H2)	no	$\{H1, H2\} - \emptyset$	$\emptyset - \emptyset$
append(H1,H2,H3)	r_d	$\{H3\} - \emptyset$	$\{H1^{(\cdot,1)}, H2\} - \{H1^{(\cdot,1)}, H2, H3\}$
merge(H1,H2,H3)	no*	$\{H3\} - \emptyset$	$\{H1, H2\} - \{(H1, H3), (H2, H3)\}$
merge_sort(H1,H2)	r_{d+i}	$\{H2\} - \emptyset$	$\{H1^{(\cdot,1)}\} - \{(H1^{(\cdot,1)}, H2^{(\cdot,1)})\}$
length(H1,H2)	no	$\{H1, H2\} - \emptyset$	$\emptyset - \emptyset$
length_2(H1,H2,H3)	no	$\{H1, H3\} - \emptyset$	$\{H2\} - \{(H2, H3)\}$
split_list(H1,H2,H3,H4)	r_d	$\{H3, H4\} - \emptyset$	$\{H2\} -$ $\{(H4, H2), (H3^{(\cdot,1)}, H2^{(\cdot,1)})\}$
remove_adjacent_dups(H1,H2)	r_i	$\{H2\} - \emptyset$	$\{H1^{(\cdot,1)}\} - \{(H1^{(\cdot,1)}, H2^{(\cdot,1)})\}$
remove_adjacent_dups_2(H1,H2,H3)	r_d	$\{H3\} - \emptyset$	$\{H1^{(\cdot,1)}, H2\} -$ $\{(H3^{(\cdot,1)}, H2), (H3^{(\cdot,1)}, H1^{(\cdot,1)})\}$

Table 1: Overview of the analysed predicates. r_d = direct reuse, r_i = indirect reuse, r_{d+i} = combined reuse, no = no reuse possible, no^* = missed reuse.

RAM, using SunOS Release 5.7, under a usual workload. The Prolog-engine used was Master Prolog, release 4.1 ERP. On this platform the analysis of `mylabelopt_main` within `labelopt` (already converted into AMAI instructions), took in average 2.84 seconds.

In a second experiment we explicitly included all predicates of the imported module `opt_util` instead of using dummy predicates. The resulting module herewith contained about 100 predicates (which is a rare situation for normal real-life projects using modules). The analysis of `mylabelopt_main` under these conditions took more than 20 minutes. One reason for this high analysis time is to be found in the proliferation of the versions of the predicates (e.g. the analysis detects 7 different call patterns for `append/3`, which means that 7 versions are analysed, three of which allow reuse). The number of versions can be reduced when combining goal dependent and goal independent analysis (see next section). A second reason is related to the number of functors defining one single type. In our experiment, the concerned predicates mainly manipulate lists of instructions, where the instruction-type is defined in terms of 26 different functors. When starting to select substructures below those functors, an explosive growth of the number of aliases occurs. To prevent this, a widening operation will have to be developed which collapses all those aliases into one. For example a wildcard “*” in a

selector $s.*$ of a type t indicates that all type nodes $t^{s.s_1}$ are selected for all selectors s_1 .

4 Towards a module based analysis

Large software is distributed over several modules and separate compilation of modules is necessary. The assumption that the call graph between modules has a tree structure is not too restrictive. It allows to analyse modules in a bottom-up fashion, making the results of the analysis of the exported predicates of a module available for the analysis of the modules higher up in the call tree. In this section we explore how such a modular analysis can be organised.

The problems to be solved are:

- The definition of the called predicate (e.g. q) has to be analysed without knowing the context (its call pattern) in which it is called.
- When analysing the definition of a predicate (e.g. r) containing a call to q , the caller's context is likely different from the assumed context while q was analysed. However, q cannot be re-analysed.

For domains with the right properties, it is well known that the results of a goal independent analysis [9] of q can be used to compute a safe approximation of a (goal dependent) call to q in the analysis of r . As we will argue in Subsection 4.1, this is possible for our analysis, so r can be analysed without re-analysing q . Another difficulty is when the analysis of q shows that there is opportunity for reuse. Typically, this reuse will not be safe for every calling context, so it is necessary to provide two versions of q , the standard one without reuse and a variant q^r with reuse. So, there is also a need to provide information which allows the caller to test whether it can call the version with reuse or has to call the standard version. Which information has to be saved from the goal independent analysis is discussed in Subsection 4.2.

4.1 Goal independent analysis

As explained in Section 2, forward use, backward use and aliases are separated in a local and a global component. The total forward/backward use is simply the union of the local and the global component. The computation of the local component is independent of the existing global component. Moreover, procedure-exit has not to return anything about forward use, while returning only the final local backward use. Also the aliases are separated in a local and a global component. As described in [4], the full set of aliases in a program point is given by the so called *alternating* closure of both components (pairs connected by a path which alternates between a global and

a local alias). The value of the local component in a program point is independent of the global component and the local component is the only one returned by procedure-exit. Hence, a goal independent analysis returns the same aliases as a goal dependent one. So, using the results of the goal independent analysis of q for analysing a call to q results in exactly the same abstract state in the program point following the call as when performing a goal dependent analysis of that call to q .

However, there is also the issue of possible reuse inside q . Liveness in a program point depends on both the global and the local component. A truly goal independent analysis of q would start with all global components empty. Obviously, it will detect a maximal number of opportunities for reuse. However, the assumption that none of the output variables will be used after exiting the predicate is an unrealistic one. Typically, all the outputs will be used. While not using all outputs can make sense, it is definitely inefficient in time and memory as the execution of the predicate will create the unneeded output argument. It is not the purpose of our analysis to overcome such inefficiencies. A preceding analysis should trap such calls, should create a version of the predicate without the unneeded output argument and should redirect the call to that version. Hence it is much more sensible to perform an analysis under the assumption that all output arguments will be needed, i.e. to initialise the global forward use with all the head variables in output positions. In what follows we mean such an analysis when using the term goal independent analysis.

In summary, a module can be analysed by analysing its exported predicates in a goal independent way. The analysis of such a predicate starts with an empty set of global aliases, an empty set of variables in global backward use and with the head variables occurring in output positions of the predicate in global forward use. Such an analysis returns a set of head variables in local backward use and a set of local aliases between head variables. When analysing a goal dependent call in a module importing the predicate, the results of the goal independent analysis are used to compute the effect of the call on the abstract state. However, one problem remains: the goal independent analysis can detect opportunities for reuse of data structures. If so, there is no guarantee that reuse is allowed for all calls to the predicate as the liveness, which decides whether reuse is possible, depends also on the global components. Hence, if reuse is possible, the goal independent analysis has to indicate that two versions have to be made, the standard one without reuse and the optimised one with reuse. Then it also has to provide information which allows the caller to decide which version it can use.

4.2 Deciding about reuse

In what follows, a component is given a subscript p when its value depends on the program point and it is given a superscript (gi or gd) when its value differs between the goal independent and the goal dependent analysis. Let

us consider a program point p prior to a deconstruction $X \Rightarrow \dots$ in the predicate definition for q . Let LU_p be the union of the local forward and backward use in program point p and $Lalias_p$ be the local aliases at p . Let $Galias$ be the global aliases (the same in every program point of q). As described in [4], the full set of aliases is given by the alternating closure of the two sets ($Alias_p = Altclos(Galias, Lalias_p)$). In the goal independent case, the initial liveness on entry of q , $Live_0^{gi}$ is the set of head variables in output positions of q and $Galias$ is empty. The formula expressing the liveness at p [4] reduces to:

$$Live_p^{gi} = Live_0^{gi} \cup \{X^\epsilon \mid X \in LU_p\} \cup \\ \{X^{sX} \mid (X^{sX}, Y^{sY}) \in Lalias_p \wedge Y \in LU_p\} \cup \\ \left\{ X^{sX_1} \left| \begin{array}{l} (X^{sX}, Y^{sY}) \in Lalias_p \text{ and} \\ \exists s_1, s_2 \text{ such that } Y^{s_1} \in Live_0^{gi} \text{ and} \\ \text{either } s_Y \equiv (s_1.s_2) \wedge s_{X_1} \equiv s_X \\ \text{or } (s_Y.s_2) \equiv s_1 \wedge s_{X_1} \equiv (s_X.s_2) \end{array} \right. \right\}$$

In the goal dependent case we obtain:

$$Live_p^{gd} = Live_0^{gd} \cup \{X^\epsilon \mid X \in LU_p\} \cup \\ \{X^{sX} \mid (X^{sX}, Y^{sY}) \in Altclos(Galias^{gd}, Lalias_p) \wedge Y \in LU_p\} \cup \\ \left\{ X^{sX_1} \left| \begin{array}{l} (X^{sX}, Y^{sY}) \in Altclos(Galias^{gd}, Lalias_p) \text{ and} \\ \exists s_1, s_2 \text{ such that } Y^{s_1} \in Live_0^{gd} \text{ and} \\ \text{either } s_Y \equiv (s_1.s_2) \wedge s_{X_1} \equiv s_X \\ \text{or } (s_Y.s_2) \equiv s_1 \wedge s_{X_1} \equiv (s_X.s_2) \end{array} \right. \right\}$$

What we are interested in is: Given that $X^\epsilon \notin Live_p^{gi}$, what is a sufficient condition for $X^\epsilon \notin Live_p^{gd}$? A brute force approach is to make $Lalias_p$ and LU_p available for the callers of q . Then the caller can compute $Live_p^{gd}$ from scratch and check whether X^ϵ belongs to it. The definition of q can have many local variables and $Lalias_p$ can be large; so it can be a rather expensive computation. So, let us analyse whether the amount of information to be saved can be reduced.

Comparing both formulas and given that $X^\epsilon \notin Live_p^{gi}$, we can observe that $X^\epsilon \in Live_p^{gd}$ only if one of the following conditions holds:

1. $X^\epsilon \in Live_0^{gd}$
2. $(X^\epsilon, Y^s) \in Altclos(Galias^{gd}, Lalias_p) \wedge Y \in LU_p$
3. $(X^\epsilon, Y^{sY.s}) \in Altclos(Galias^{gd}, Lalias_p) \wedge Y^{sY} \in Live_0^{gd}$

To check condition 1 during the goal dependent analysis, we only need to know from the goal independent analysis which variable can be reused by it (to be stored as value for $reuse_q$). Note that condition 1 can prevent reuse only if X is an input head variable because only head variables occur in $Live_0^{gd}$ and output head variables cannot be reused as they occur in $Live_0^{gi}$ and thus in $Live_p^{gi}$.

A first observation about the other two conditions is that $Galias^{gd}$ only contains aliases between input head variables and $Lalias_p$ cannot contain aliases between input head variables. Indeed, Mercury is such that input head variables cannot be further instantiated, hence no new aliases can be created between them. For what concerns condition 2, if $(X^\epsilon, Y^s) \in Altclos(Galias^{gd}, Lalias_p)$ and $Y \in LU_p$ while $X^\epsilon \notin Live_p^{gi}$ then $(X^\epsilon, Y^s) \notin Lalias_p$, so we must have one of the following cases:

- (a) X and Y are input head variables and $(X^\epsilon, Y^s) \in Galias^{gd}$ and $Y^s \in Live_p^{gi}$ (because $Y \in LU_p$).
- (b) X is an input head variable and Y is a local variable and there is a head variable $Hvar$ such that $(X^\epsilon, Hvar^{s_1}) \in Galias^{gd}$ and $(Hvar^{s_2}, Y^{s_3}) \in Lalias_p$ and $Y \in LU_p$ for appropriate selectors¹ s_1, s_2 , and s_3 . However, then also $Hvar^{s_2} \in Live_p^{gi}$, hence this case is covered by case (a) because $(X^\epsilon, Hvar^s) \in Galias^{gd}$ and $Hvar^s \in Live_p^{gi}$ for an appropriate value of s .
- (c) X is a local variable, $Hvar$ and Y are input head variables, $(Hvar^{s_1}, Y^{s_2}) \in Galias^{gd}$, $Y^{s_2} \in Live_p^{gi}$ (because $Y \in LU_p$), and $(X^\epsilon, Hvar^{s_3}) \in Lalias_p$ for appropriate selectors s_1, s_2 , and s_3 .
- (d) X and Y are local variables, $Hvar1$ and $Hvar2$ are input head variables, $(X^\epsilon, Hvar1^{s_1}) \in Lalias_p$, $(Hvar2^{s_2}, Y^{s_3}) \in Lalias_p$, $(Hvar1^{s_4}, Hvar2^{s_5}) \in Galias^{gd}$ and $Y \in LU_p$ for appropriate selectors s_1, s_2, s_3, s_4 , and s_5 . Similarly as in case (b), also $Hvar2^{s_2} \in Live_p^{gi}$ hence the case is covered by case (c).

We are left with the cases (a) and (c). We observe that the information from the goal independent analysis which is needed to check the second condition for reuse is:

- The liveness of the input head variables in $Live_p^{gi}$.
- The aliases in $Lalias_p$ between X^ϵ and input head variables.

For what concerns condition 3, Y must be a head variable as $Y^{s_Y.s} \in Live_0^{gd}$. If $(X^\epsilon, Y^{s_Y.s}) \in Altclos(Galias^{gd}, Lalias_p)$ then either $(X^\epsilon, Y^{s_Y.s})$ belongs to $Lalias_p$ or to $Galias^{gd}$ or there is an input head variable $Hvar$

¹Do not worry about which selectors are appropriate. Our interest is only in identifying the elements of $Alias_p$ which are needed to decide the condition. The function $Altclos$ is doing the tedious job of finding the right values for the selectors.

and $(X^\epsilon, Hvar^{s_1}) \in Lalias_p$ and $(Hvar^{s_2}, Y^{s_3}) \in Galias^{gd}$ for appropriate selectors s_1, s_2 , and s_3 . We observe that the information from the goal independent analysis which is needed to check the third condition for reuse consist of :

- The aliases between X^ϵ and head variables in $Lalias_p$

Hence to check whether a call to q can use the version q^r with reuse of X , it suffices that we save from the goal independent analysis:

- $reuse_q$, the name of the variable X which can be reused.
- $Plive_q^{gi}$, the projection on the input head variables of q of the liveness in the program point preceding the reuse.
- $Palias_q^{gi}$, the pairs $(X^\epsilon, Hvar^s)$ with $Hvar$ a head variable, of the local aliases in the program point preceding the deconstruction.

When the goal independent analysis of a predicate r which calls q shows that q^r can be used then a version r^r of r using q^r has to be created and information to decide whether r^r can be used is needed. This information is computed in a similar way. Let $Lalias_{p_0}$ and $Live_{p_0}^{gi}$ be the information in the program point of r which precedes the call to q and let ρ be a renaming which maps the formal arguments in the head of q to the actual arguments in the call to q . Then the three components are:

- $reuse_r = reuse_q\rho$.
- $Plive_r^{gi} =$ the projection on the input head variables of r of $Live_{p_0}^{gi} \cup Plive_q^{gi}\rho$.
- $Palias_r^{gi} =$ the pairs $(X^\epsilon, Hvar^s)$ from $Altclos(Lalias_{p_0}, Palias_q^{gi})\rho$ with $Hvar$ a head variable of r and X the reused variable.

The next subsection will illustrate this with an example.

4.3 Example

Consider the predicate `eliminate(I,List)` (figure 4); where I is of type $instruction == (instr - string)$ and $List$ is of type $list(instruction)$. Its goal independent analysis starts with $Live_0 = \{List^\epsilon\}$. The goal independent analysis yields the the alias $(I^{(-,2)}, List^{(.,1).(-,2)})$. The analysis also shows that I^ϵ is not live in the program point preceding the deconstruction $I \Rightarrow Instr0 - Comment0$. As there is a subsequent construction $NewI \Leftarrow NewInstr - Comment0$, reuse is possible in the goal independent case and the information needed to choose between `eliminate` and `eliminater` is needed. We have:

- $reuse_{eliminate} = I$

- $Plive_{\text{eliminate}}^{gi} = \emptyset$ (no live input variables in the program point).
- $Palias_{\text{eliminate}}^{gi} = \emptyset$ (no aliases in the program point).

Now let us consider the goal independent analysis of `instr_list`. During the analysis, a goal dependent call `eliminate(I0,R)` occurs. Applying the predicate-entry operation on the call results in $Live_0^{gd} = \{I^{(-,1)}\}$ and $Galias^{gd} = \emptyset$. One can check that $I^\epsilon \notin Live_0^{gd}$ and $(I^\epsilon, Y^{s0.s1}) \notin Altclos(Galias^{gd}, Palias_{\text{eliminate}}^{gi})$ for any $Y^{s0} \in Plive_{\text{eliminate}}^{gi}$; `eliminater` can therefore be used and a `instr_listr` can be generated.

Finally we show the derivation of the information needed to decide on the use of `instr_listr(L1,L2)`. $\rho = \{I \rightarrow I0, List \rightarrow R\}$. Considering the program point before the call to `eliminate`, we obtain:

- $reuse_{\text{instr_list}} = I0$
- $Plive_{\text{instr_list}}^{gi} = \{L1^\epsilon\}$ (because $L1^\epsilon$ is an alias of *MoreL* which is live).
- $Palias_{\text{instr_list}}^{gi} = \{(I0^\epsilon, L1^{(\cdot,1)})\}$.

Because of $L1^\epsilon$ being live and of the alias $(I0^\epsilon, L1^{(\cdot,1)})$, the condition for using `instr_listr(L1,L2)` cannot be satisfied. However, our goal dependent analysis shows that reuse is possible. The loss of precision is (again) due to the fact that the recursive list selector $(.,2)$ is equivalent to the empty selector ϵ . If they were distinguished, then $Plive_{\text{instr_list}}^{gi} = \{L1^{(\cdot,2)}\}$ and the condition for the use of `instr_listr(L1,L2)` could be met by callers of the predicate.

This shows that goal independent analysis can be less precise than goal dependent analysis. To obtain the best precision, goal independent analysis should be restricted to the predicates a module is exporting. On the other hand, goal independent analysis can save a lot of work. For example, in an experiment where the complete original code of the module *label_opt* was analysed, `append/3` was analysed for 7 different call patterns (3 of them allowed reuse). Analysing all predicates in a goal independent way, each predicate has to be analysed only once. As the current handling of recursive selectors also was at the source of the lack of precision in analysing the `merge/3` predicate, it is definitely worthwhile to improve that aspect of our analysis. The increase in complexity should be more than compensated by the adoption of goal independent analysis for all predicates.

As a last remark, there are other sources of reuse in `instr_list/2`: there is a recursive call and there is a call to `append/3`. For each of them a triplet describing the information needed to check for reuse can be derived. This could lead to a proliferation of versions, (`eliminater` combined with `appendr`, `eliminater` combined with `append`, ...). The drastic solution of having only two versions, one which combines all possible reuse (having three triplets that have to be satisfied by the caller) and one without any reuse seems to be adequate for the module under consideration (as far as we can check by hand).

5 Conclusion

We have reported some initial experiments with live structure analysis for Mercury. The analysis of a module of the compiler revealed that most opportunities for reuse are detected. Some opportunities for reuse are unused due to the imprecision in the handling of recursive data structures. One solution to overcome this problem is to modify the Mercury HLDS code, delaying the selection of a recursive component of the data structure until the component is to be used in the next instruction. Another plausible solution is to refine the representation of aliases over recursive data structures. This would offer the additional benefit of increasing the precision of the goal independent analysis. Goal independent analysis is necessary for the predicates a module exports. In addition, its generalised use would also substantially reduce the cost of the analysis.

With separate compilation of modules, the analysis – when integrated in the compiler – would lead to two versions² for exported predicates: a version with reuse of data structures, and a version without. The analysis would also generate the information needed by callers of a predicate to perform their own live structure analysis and to decide which version to call.

We propose to insert `reuse(X)` pragma's in the HLDS code to indicate to the compiler that the toplevel of `X` is available for reuse. We could also derive “destructive-input/unique-output” modes (not described in the paper due to lack of space) but then the compiler would have to redo part of our analysis to detect where exactly reuse becomes possible.

Analysis time for the code shown in the appendix is quite acceptable. However, the analysis time for the complete original module (and its imported modules) is not. These results are obtained with a very early prototype, we expect that substantial improvements are feasible in the near future. We also plan to implement the automated derivation of the reuse pragma, so that more experiments can be done. Other future work is experimentation with goal independent analysis (to better understand its effect on efficiency and precision) and revision of our abstract domain to obtain a better handling of aliases over recursive data structures. Also the widening operator mentioned in Section 3.3 will be developed.

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.

²Only one when reuse is always or never possible

- [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] Dean Jacobs and Anno Langen. Static analysis of logic programs for independent AND-parallelism. *Journal of Logic Programming*, 13(2 &3):291–314, May/July 1992.
- [10] Gerda Janssens, Maurice Bruynooghe, and Veroniek Dumortier. A blueprint for an abstract machine for abstract interpretation of (constraint) logic programs. In J.W. LLoyd, editor, *Logic Programming, Proceedings of the 1995 International Symposium (ILPS'95)*, pages 336–350, Portland, Oregon, 1995. MIT Press.
- [11] 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.
- [12] 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.
- [13] 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.
- [14] 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.

- [15] 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.
- [16] 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.
- [17] Mads Tofte, Lars Birkedal, Martin Elsmann, Niels Hallenberg, Tommy Højfeldt Olesen Højfeldt, 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, Albuquerque, 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.

A Full listing of *labelopt*

Note: we intend to make the appendix available on-line (WWW) as it will not be possible to include it in the final version.

```

%-----%
% Copyright (C) 1994-1997 The University of Melbourne.
% This file may only be copied under the terms of the GNU General
% Public License - see the file COPYING in the Mercury distribution.
%-----%

% labelopt.m - module to eliminate useless labels and dead code.

% Author: zs.
% K.U.Leuven: explicit inclusion of list-manipulating predicates,
%             opt-util-related predicates substituted by dummies.
%-----%

:- module mylabelopt.
```

```

:- interface.

:- import_module bool, list.
:- import_module llds.

    % Build up a set showing which labels are branched to,
    % then traverse the instruction list removing unnecessary labels.
    % If the instruction before the label branches away, we also
    % remove the instruction block following the label.

:- pred mylabelopt_main(list(instruction), bool, list(instruction), bool).
:- mode mylabelopt_main(in, in, out, out) is det.

    % Build up a set showing which labels are branched to.

:- pred mylabelopt__build_useset(list(instruction), set(label)).
:- mode mylabelopt__build_useset(in, out) is det.

%-----%

:- implementation.

:- import_module opt_util.
:- import_module std_util.

mylabelopt_main(Instrs0, Final, Instrs, Mod) :-
    mylabelopt__build_useset(Instrs0, Useset),
    mylabelopt__instr_list(Instrs0, yes, Useset, Instrs1, Mod),
    ( Final = yes, Mod = yes ->
        mylabelopt_main(Instrs1, Final, Instrs, _)
    ;
        Instrs = Instrs1
    ).

%-----%

mylabelopt__build_useset(Instrs, Useset) :-
    set_init(Useset0),
    mylabelopt__build_useset_2(Instrs, Useset0, Useset).

:- pred mylabelopt__build_useset_2(list(instruction), set(label), set(label)).
:- mode mylabelopt__build_useset_2(in, in, out) is det.

mylabelopt__build_useset_2([], Useset, Useset).
mylabelopt__build_useset_2([Instr | Instructions], Useset0, Useset) :-
    Instr = Uinstr - _Comment,
    opt_util_instr_labels(Uinstr, Labels, _CodeAddresses),
    set_insert_list(Useset0, Labels, Useset1),
    mylabelopt__build_useset_2(Instructions, Useset1, Useset).

%-----%

    % Go through the given instruction sequence. When we find a label,
    % we check whether the label can be branched to either from within
    % the procedure or from the outside. If yes, we leave it alone.

```

```

    % If not, we delete it. We delete the following code as well if
    % the label was preceded by code that cannot fall through.

:- pred mylabelopt_instr_list(list(instruction), bool, set(label),
    list(instruction), bool).
:- mode mylabelopt_instr_list(in, in, in, out, out) is det.

mylabelopt_instr_list([], _Fallthrough, _Useset, [], no).
mylabelopt_instr_list([Instr0 | MoreInstrs0],
    Fallthrough, Useset, MoreInstrs, Mod) :-
    Instr0 = Uinstr0 - _Comment,
    ( Uinstr0 = label(Label) ->
        (
            ( Label = exported(_)
              ; Label = local(_)
              ; set_member(Label, Useset)
            )
        )
    ->
        ReplInstrs = [Instr0],
        Fallthrough1 = yes,
        Mod0 = no
    ;
        mylabelopt_eliminate(Instr0, yes(Fallthrough),
            ReplInstrs, Mod0),
        Fallthrough1 = Fallthrough
    )
;
    ( Fallthrough = yes ->
        ReplInstrs = [Instr0],
        Mod0 = no
    ;
        mylabelopt_eliminate(Instr0, no, ReplInstrs, Mod0)
    ),
    opt_util_can_instr_fall_through(Uinstr0, Canfallthrough),
    ( Canfallthrough = yes ->
        Fallthrough1 = Fallthrough
    ;
        Fallthrough1 = no
    )
),
mylabelopt_instr_list(MoreInstrs0, Fallthrough1, Useset,
    MoreInstrs1, Mod1),
list_append(ReplInstrs, MoreInstrs1, MoreInstrs),
( Mod0 = no, Mod1 = no ->
    Mod = no
;
    Mod = yes
).

% Instead of removing eliminated instructions from the instruction list,
% we can replace them by placeholder comments. The original comment
% field on the instruction is often enough to deduce what the
% eliminated instruction was.

:- pred mylabelopt_eliminate(instruction, maybe(bool), list(instruction), bool).

```

```

:= mode mylabelopt__eliminate(in, in, out, out) is det.

mylabelopt__eliminate(Uinstr0 - Comment0, Label, Instr, Mod) :-
  labelopt_eliminate_total(Total),
  (
    Total = yes,
    Instr = [],
    Mod = yes
  ;
    Total = no,
    ( Uinstr0 = comment(_) ->
      Comment = Comment0,
      Uinstr = Uinstr0,
      Mod = no
    ;
      ( Label = yes(Follow) ->
        ( Follow = yes ->
          Uinstr = comment("eliminated label only")
        ;
          % Follow = no,
          Uinstr = comment("eliminated label and block")
        )
      ;
      % Label = no,
      Uinstr = comment("eliminated instruction")
    ),
    Comment = Comment0,
    Mod = yes
  ),
  Instr = [Uinstr - Comment]
).

:= pred labelopt_eliminate_total(bool).
:= mode labelopt_eliminate_total(out) is det.

labelopt_eliminate_total(yes).

%-----%

% opt_util-related definitions

% dummy
:= pred opt_util_instr_labels(instr,list(label),list(code_addr)).
:= mode opt_util_instr_labels(in, out, out) is det.

opt_util_instr_labels(_,[],[]).

% dummy
:= pred opt_util_can_instr_fall_through(instr, bool).
:= mode opt_util_can_instr_fall_through(in, out) is det.

opt_util_can_instr_fall_through(_,yes).

% set-related definitions

```

```

    % set(T) == set_ordlist(T) == list(T).
:- type set(T) == list(T).

:- pred set_init(set(T)).
:- mode set_init(out) is det.

set_init([]).

:- pred set_insert_list(set(T),list(T),set(T)).
:- mode set_insert_list(in,in,out) is det.

set_insert_list(Set0,List0,Set):-
    list_sort_and_remove_dups(List0,List),
    list_merge_and_remove_dups(List,Set0,Set).

:- pred set_member(T,set(T)).
:- mode set_member(in,in) is semidet.

set_member(T,L):- list_member(T,L).

% list-related definitions

:- pred list_append(list(T),list(T),list(T)).
:- mode list_append(in,in,out) is det.

list_append([],Y,Y).
list_append([X|Xs],Y,[X|Zs]):-
    list_append(Xs,Y,Zs).

:- pred list_sort_and_remove_dups(list(T),list(T)).
:- mode list_sort_and_remove_dups(in,out) is det.

list_sort_and_remove_dups(L0, L) :-
    list_merge_sort(L0, L1),
    list_remove_adjacent_dups(L1, L).

:- pred list_merge_sort(list(T),list(T)).
:- mode list_merge_sort(in,out) is det.

list_merge_sort([], []).
list_merge_sort([X], [X]).
list_merge_sort(List, SortedList) :-
    List = [_,_],
    list_length(List, Length),
    HalfLength is Length // 2,
    ( list_split_list(HalfLength, List, Front, Back) ->
        list_merge_sort(Front, SortedFront),
        list_merge_sort(Back, SortedBack),
        list_merge(SortedFront, SortedBack, SortedList)
    );
    error("list__merge_sort").

:- pred list_length(list(T),int).

```

```

:- mode list_length(in,out) is det.

list_length(L, N) :-
    list_length_2(L, 0, N).

:- pred list_length_2(list(T), int, int).
:- mode list_length_2(in, in, out) is det.

list_length_2([], N, N).
list_length_2([- | L1], N0, N) :-
    N1 is N0 + 1,
    list_length_2(L1, N1, N).

:- pred list_split_list(int, list(T), list(T), list(T)).
:- mode list_split_list(in, in, out, out) is semidet.

list_split_list(N, List, Start, End) :-
    ( N = 0 ->
        Start = [],
        End = List
    ;
        N > 0,
        N1 is N - 1,
        N1 = 1,
        List = [Head | List1],
        Start = [Head | Start1],
        list_split_list(N1, List1, Start1, End)
    ).

:- pred list_merge(list(T), list(T), list(T)).
:- mode list_merge(in, in, out) is det.

list_merge(A, B, C) :-
    ( A = [X|Xs] ->
        ( B = [Y|Ys] ->
            C = [Z|Zs],
            (
                compare(<, X, Y)
            ->
                Z = X,
                list_merge(Xs, B, Zs)
            ;
                Z = Y,
                list_merge(A, Ys, Zs)
            )
        ;
            C = A
        )
    ;
        C = B
    ).

:- pred list_remove_adjacent_dups(list(T),list(T)).
:- mode list_remove_adjacent_dups(in,out) is det.

```

```

list_remove_adjacent_dups([], []).
list_remove_adjacent_dups([X|Xs], L) :-
    list_remove_adjacent_dups_2(Xs, X, L).

:- pred list_remove_adjacent_dups_2(list(T), T, list(T)).
:- mode list_remove_adjacent_dups_2(in, in, out) is det.

list_remove_adjacent_dups_2([], X, [X]).
list_remove_adjacent_dups_2([X1|Xs], X0, L) :-
    (
        X0 = X1
    ->
        list_remove_adjacent_dups_2(Xs, X1, L)
    ;
        L = [X0 | L0],
        list_remove_adjacent_dups_2(Xs, X1, L0)
    ).

:- pred list_merge_and_remove_dups(list(T),list(T),list(T)).
:- mode list_merge_and_remove_dups(in,in,out) is det.

list_merge_and_remove_dups(A, B, C) :-
    list_merge(A,B,C).

:- pred list_member(T,list(T)).
:- mode list_member(in,in) is semidet.

list_member(X, [X | _]).
list_member(X, [_ | Xs]) :-
    list_member(X, Xs).

```
