Automatic source code reduction*

Jiří Diviš and Ondřej Bojar

Charles University, Faculty of Mathematics and Physics
Malostranské nám. 25, Praha 1, CZ-118 00, Czech Republic
jiridivis@gmail.com, bojar@ufal.mff.cuni.cz

Abstract. *The aim of this paper is to introduce Reductor, a program that automatically removes unused parts of the source code of valid programs written in the Mercury language. Reductor implements two main kinds of reductions: statical reduction and dynamical reduction. In the statical reduction, Reductor exploits semantic analysis of the Melbourne Mercury Compiler to find routines which can be removed from the program. Dynamical reduction of routines additionally uses Mercury Deep Profiler and some sample input data for the program to remove unused contents of the program routines. Reductor modifies the sources of the program in a way, which keeps the formatting of the original program source so that the reduced code is further editable.*

1 Introduction

Mercury [1] is a fast logic and functional programming language with advanced error detection features, developed for writing large real-world programs. Its syntax builds upon Prolog syntax (Prolog predicate clauses), adding some new declarations. These declarations are used for error checking and to speed-up the execution of a compiled program. Another great feature of Mercury is that it can compile to C or Java and thus it easily interfaces with foreign code. Motivations behind the design of Mercury are very well summed up in [2].

Reductor [3] is mainly intended for programmers who wish to understand and/or reuse code of a big program or start an independent project based on just a few features of an existing one. This is why Reductor modifies the sources of the program in a way, which preserves the original formatting, so that the reduced code is further editable. But there are also other uses, like reduction of the size of executables, decreasing the compilation time of reduced programs or releasing only a subsection of a huge project.

We suggest the reader gets acquainted with basics of Mercury language by reading one of the following summaries: Ralph Becket's tutorial [4] and Seriál Mercury on ROOT.CZ [5] (in Czech).

* This work has been supported by the grants EuroMatrixPlus (FP7-ICT-2007-3-231720 of the EU and 7E09003 of the Czech Republic) and MSM 0021620838.

1.1 Features of Reductor

Reductor implements two different kinds of reductions — statical reduction and dynamical reduction. The user chooses between the statical and dynamical reduction, not both at the same time. There is also a trivial module reduction, which is done implicitly.

The **module reduction** takes all modules from the current directory that are (transitively) imported by the main module of the program and copies them without any change in their contents into specified destination directory.

The **statical reduction** takes as an input a Mercury program and removes some inaccessible routines from the sources of the program. Inaccessible routines are the routines that program would never call and that can be deleted, because they do not appear in any definition of any of the remaining routines.

The **dynamical reduction** removes parts of routines that were never used on several inputs to the program. The reduction is based on profiling data from the runs of the program on some specified input. The deep profile is generated by Mercury Deep Profiler.

The dynamically reduced program can be compiled but Reductor guarantees that the reduced program will generate the same output only for inputs the deep profile was obtained on.[1] On different inputs, the reduced program may terminate with an exception.

Reductor tries to preserve the original formatting of the original program to as much as possible. Code segments corresponding to goals or routines of the program to be removed are commented out using `/*red: ... :red*/`, any comments of the kind `/* ... */` in such a segment are transformed into `/nested:* ... *: nested/`. If Reductor needs to add some code, then it is done by delimiting the inserted term by two newlines and a comment indicating that the term was inserted by Reductor.

2 Overview of Reductor

Reductor consists of the following parts.

[1] We assume that the output of the original program is determined solely by its input data.

1. *Intermodule Representation of the Program (IMR).* This structure represents the entire Mercury program on various levels of representation (see Section 2.1). It is used by all three remaining parts of Reductor and it is based on the results of the semantic analysis of Melbourne Mercury Compiler (MMC) which we reuse and modify to suit our needs.
2. *I/O Interface.* (Section 3) This part handles the representation of the changes in the program on the term level (see below) and offers an interface for making the changes proposed by the statical and dynamical reduction modules. It also outputs the reduced sources.

   This representation is designed for storing the source changes while preserving the original source and its formatting as much as possible.
3. *Dynamical Reduction.* (Section 4) Using the data from IMR, this module dynamically reduces the program and submits the changes to the I/O interface. This module also extracts the data from the deep profile.
4. *Statical Reduction.* (Section 5) Using the data from IMR, this module statically reduces the program and submits the changes to the I/O interface.

2.1 Levels of representation of a module

Reductor utilizes the results of the semantic analysis of Melbourne Mercury Compiler (MMC) [1]. We thus extracted the part of MMC code responsible for the compilation up to the end of semantic analysis and we based the Reductor on the extracted code.

The mentioned part of the compilation of a module consists of the following four main stages. At these stages, the program is represented in certain data structures. We call these data structures the level of representation of the program.

Mercury program is composed of declarations and clauses, we will use the term **item** for either those. The *first* stage is lexical analysis, after which a module is represented as a list of lists of tokens. Each such list of tokens corresponds to an item. We call this representation **Token-Level Representation (ToLR)**.

Each item in a program corresponds to what can be considered the standard ISO Prolog **term** for the purposes of this paper. The *second* stage consists of parsing of the tokens into terms. After this stage a module is represented as a list of terms. We will refer to this as the **Term-Level Representation (TeLR)**. The parts of a term that are themselves terms may be called **subterms** and the term that corresponds to an item may be sometimes called **base term** to avoid confusion.

After the *third* stage, each term is converted into a syntax tree of the item.[2] We will call this the **Item-Level Representation (ILR)** of a program. After this stage, each item is categorized based on the information extracted solely from its corresponding term(s), not from other terms of the module. The parse tree still contains most of the syntactic details of the item in the program.

Finally, after the *fourth* stage, the data structure called by the designers of MMC the *High Level Data Structure (HLDS)* is constructed and filled with the results of the semantic analysis of a module. This includes the following: Declarations from imported modules are added, predicates and their goals and subgoals are annotated with inferred determinisms and modes, variables in scopes of each subgoal of each predicate are annotated with their modes and types, goals are reduced to a certain equivalent subset of the original goals from the code and thus lot of syntactic detail irrelevant to the semantic analysis of the module source is lost.

After this stage, still most of the items from the source have their corresponding structures in HLDS, but not vice-versa—e.g. there are structures for predicates that were automatically generated. We call the described state of HLDS the **High-level Representation (HLR)**.

As a part of the fourth stage, based on the mode analysis of the program, a **procedure** is constructed for each mode of the predicate/function because clauses have to be reordered for each mode independently.

Also, in HLR, all clauses of a single procedure are combined into one goal. Similarly as for term, we define the **subgoal** and the **top-level goal** (base goal).

From now on, we will usually ignore the difference between functions and predicates calling them simply **predicates**.

Finally, we define the **call site** as a goal which can call a procedure (this includes goals that call lambda expressions).

2.2 Inter-module representation

Considering the program representation, there is a big difference between the needs of Reductor and the needs of Compiler in the way modules are handled. Compiler compiles each module of the program independently but Reductor's statical reduction needs to gather semantic analysis for all modules at once and store it for later uses. This fact has also some implication on memory requirements of Reductor.

Our approach to combining MMC's data structures of each module into IMR is quite simple. We use

[2] Some terms may be combined to form as single item.

map data types to map fully qualified module names to these structures. There are also a few structures which are combined in a more sophisticated way to improve efficiency, but these details are unnecessary for the purposes of this paper.

From the beginning, it was clear that we will have to reuse some of the MMC source. The question was whether to use only ILR or use also HLR of the program. We decided to use MMC's semantic analysis in order to achieve powerful enough reductions, especially in the case of dynamical reduction. On the other hand, the use of HLR makes Reductor more dependent on MMC and makes it harder to understand, which is among other things caused by the fact that MMC is designed for other purposes than simplicity and reusability of the semantic analysis alone.

As a part of IMR, there is also a structure that stores information about the module dependencies (imports) and the source files that they came from. This structure is extracted from the `make`-like part of MMC that directs compilation.

3 I/O interface

The I/O interface allows both the statical and dynamical reductions to specify changes on the term level by adding, deleting or changing a particular term that corresponds to an item of a module. Each change is specified in a data type called **term change**, see below.

The I/O interface performs the following steps:

1. For each module in a program, the TeLR and a corresponding **modified ToLR** (MToLR) is constructed. MToLR extends ToLR by storing the exact position of the first character of the token in the corresponding source file.
2. A blank structure to hold term changes is set up for each module.
3. At this point, changes are submitted by the reductions. The I/O interface provides the association of items in TeLR with items in ILR, but the term change itself has to by provided by the reduction.
4. From the collected changes, we construct a **change list** for each module of the program, which represents the changes to be made in the sources in terms of list of string insertions and deletions. The algorithm for this is explained in Section 3.2.
   Each element of the change list represents either the range of positions in the sources which is to be removed or a string and the position where to insert it.
5. We copy the original sources and apply the changes specified by change lists.

3.1 Specifying the changes in a term

The changes in a term are represented by a term change. This structure mirrors the original term structure and stores, for each subterm of the term, how the subterm is changed. The change can be (1) replacing the term with the string, (2) enclosing the subterm by two strings or (3) replacing the term with one of its subterms.

Formally, the term change can have one of the following values:

no_change: The corresponding term and all of its subterms are not modified.

no_lvl_change(list(term_change)): The term is not modified at the level of its functor, the `term_change` list stores information about the subterms.

subst_ins(string, list(term_change), string): The term is to be preceded by the `string` given as the first parameter and followed by the `string` in the third parameter. The second argument represents the changes made on the term's arguments.

replace(string): The corresponding term is to be replaced by the given string.

subst_del(list(term_change)): The substructures of this term change, i.e. the lit of term changes have restricted set of allowed constructors here. There can be only `no_change` constructors, except that one `orig` constructor has to be present. The substructures of the `orig` constructor have no restrictions.
The term that corresponds to this term change is substituted by the term that corresponds to the term change with the orig constructor (which may be changed further).

orig(list(term_change)): This is applicable only if contained in substructure with `subst_del` constructor as described.

3.2 Construction of change lists

For each source file we need to construct a change list which specifies what will be modified in the file. This is done by constructing the change list for each term in the same order as they appear in the module, and concatenating the change lists. We briefly describe our approach to the problem of constructing the change list for a term.

We observed that each subterm consists of contiguous block of tokens in the MToLR. The block of tokens is not affected by any subterms other than its own.

Our algorithm is based on a synchronous traversal of a base term and its term change and identifying the lists of consecutive tokens that correspond to the subterms. This gives us for each term its beginning

and end positions in the list of tokens. From this we compute the subterm's beginning and end positions in the source file.

We determine the list of tokens that correspond to each subterm by finding how a given base term can be written using tokens, while matching it against the actual list of tokens in the MToLR. This gives us the beginning and ending token of each subterm.

While we traverse the subterm and the corresponding term change in the mentioned way, we additionally mark (according to the term change) which segments of token lists are to be removed and at which positions to insert the strings given in the term change. The change list of the term is then constructed from the position information associated with the tokens.

4 Dynamical reduction

The idea of the dynamical reduction is to take the data collected by Mercury Deep Profiler (MDP) [1] on multiple runs of the program with various input data (we will call them **test data** in the following) and based on the profile, remove most of unused code. This is achieved by cleverly substituting goals of unused branches of code with calls to the `throw` predicate (which throws an exception) so that we preserve the syntactic correctness of the program.

We will illustrate the basic idea on a simplified model. Imagine a simplified version of the execution model, which executes goals in top-down, left-to-right order. In this model we collect the information from the profile, which indicates for each atomic goal whether it was called during the execution of the runs on the test data. We then reduce each procedure by substituting topmost subgoals that do not contain any called subgoals with a call to the `throw` predicate. The resulting program will have same outputs for the runs of input data used to construct the profiling data. On different data the program may exit with an exception.

The problem with this description is that MMC does not create programs that execute goals in this simple top-down, left-to-right sequence. For example, MMC reorders conjuncts to satisfy mode declarations of the calls in the conjunction. However these issues do not prohibit a similar approach to the one described.

4.1 Overview of dynamical reduction

We present the main steps Reductor goes through to dynamically reduce a program. These steps and additional details are then discussed further in the sections below:

1. *Load the profile created on the test data:* We integrated MDP into the Reductor and we use it to collect and read the profiling data.
2. *Determine which goals were called in HLR:* (Section 4.4) We collect the data from MDP for some of the call sites in the program. The data we collect tells us which ports of the standard box model were used on those call sites. We then use the information from MMC semantic analysis to improve this information, as the data transfer from MDP to HLR is inaccurate.
3. *Determine which predicates will be reduced:* (Sections 4.2 and 4.3) Any procedures may be excluded from the reduction, if necessary, because our dynamical reduction has a local character—the reductions made in a predicate do not depend on reductions made on any other predicates. Currently, predicates that have multiple modes and typeclass methods are unsupported and thus we do not attempt to reduce them. As it will be seen later, the reduction of multi-moded predicates can be achieved by trivially combining the data of their individual procedures in HLR.
4. *Transfer the data about called atomic goals into the corresponding clauses in ILR.*
5. *Reduce the goals of an item:* Identifying the subgoals of the processed item clause that are to be substituted with an exception.
6. *Transfer the changes from ILR to TLR:* (Section 4.5) Construct the term change for the term that corresponds to the item being processed and submit the information to the I/O interface described in Section 3.

4.2 Finding goals substitutable with exceptions

This algorithm for computing removable goals is based on the observation that in Mercury, the programmer can substitute any goal with a call to `throw` predicate without compromising the syntactic correctness of the program, i.e. the resulting program compiles, but it may give lot of warnings (e.g. about the presence of singleton variables, various determinism warnings).

The algorithm processes goals of clauses in the ILR to identify removable goals in program clauses. By **removable goals**, we mean some subset of goals that can, but does not have to, be substituted with an exception, while preserving the program semantics on the test data. By **non-removable goals**, we mean the complement of the subset. We describe the algorithm here and we discuss its correctness in Section 4.3.

As an input to this algorithm, we assume that we are given the information about which atomic goals were called on the test data. More precisely, we assume that we get a superset of the called atomic goals. We define **atomic goals** as the goals that do not have any subgoals, i.e. calls and unifications.

If we consider the ordering of goals that is given by traversing the goal tree in top-down, left-to-right order, we mark all goals that are preceded by any called atomic goal (called goals included) as non-removable.

This means that some goals might not contain any called goals and still be non-removable. The reason for this is the fact that if we substituted those goals with a call to `throw`, we would change the way the goal is reordered and this could cause premature termination of the reduced program by an exception. Code reordering was the major challenge in the design of the dynamical reduction and we discuss it in Section 4.3.

The algorithm traverses the goal in the opposite order to the one described earlier. Each goal gets annotated with information saying if it is non-removable. We call this information **removal status**:

All goals that contain a single subgoal: The removal status is the same as the removal status of its subgoal.

'and', 'implication', 'equivalence': If the second-argument goal is non-removable, then the first-argument goal and all its subgoals are considered non-removable.

'or': Removable if both of its subgoals are removable, otherwise non-removable.

'if then else': The 'if then else' goal inherits removal status from its 'if' subgoal.

Atomic goals: They are non-removable if they are called, otherwise they are removable. This can be overridden with in 'and', 'implication', 'equivalence', 'or'.

4.3 Issues with code reordering

In this section, we discuss the problems associated with the fact that MMC may reorder goals. We discuss our assumptions about the compilation model of MMC and conditions that need to be satisfied for the algorithm for computing removable goals, introduced in Section 4.2, to be correct.

We note that by default, MMC reorders conjunctions to satisfy mode declarations of the calls in the conjunctions. Additionally, MMC may reorder disjunctions and optimize away some calls and do other optimizations as discussed in [6].

In a nutshell, in order for algorithm from Section 4.2 to work correctly, we need to assure that in both the original program, compiled to produce deep profile, and the reduced program, the executed goals are identical in both programs and that they are executed in the same order (all with respect to particular input data).

Strict sequential operational semantics. For the dynamical reduction we need the following three assumptions. We believe that using *strict sequential operational semantics* in MMC (explained in [6]) ensures them.

1. *No calls are optimized away.* Our implementation of dynamical reduction assumes that the data from the deep profile are accurate in the "semantic" sense. If some calls were optimized away, the collected profile would indicate that the goal was not called, which then might cause the goal to be considered removable, and as such it could be potentially substituted with an exception. This could then make the reduced program to incorrectly throw an exception on the test data.
2. *Conjunctions are reordered minimally, every time the ordering of conjuncts is the same.* This restriction will be clarified later in this section, for now we just note that the if we created two profiles on the same test data with different conjunction orderings, we may get different information about which atomic goals are called, which could again potentially lead to different output of the algorithm from Section 4.2, which can be problematic.
3. *Disjunctions are not reordered.* The reason is same as in (2).

Correctness of dynamical reduction. It is guaranteed that a program dynamically reduced for a given input will give the same results as the original one, if both are compiled with strict sequential operational semantics and the following conditions hold:

1. The conjuncts that were called in the original program have the same ordering in the reduced program with respect to each other as in the original program and the call goals call the same modes of the called predicates.
2. The originally uncalled conjuncts are not reordered in front of any originally called conjuncts.
3. In the reduced program, only originally uncalled conjuncts may be substituted with call to `throw`.

The first two conditions ensure that each procedure is called with the same inputs as in the original program. We did not determine, if the condition (1) is necessary, but we suspect that if we would not require it, there might be problems with goals with the determinism `multi`.

The failure to comply with the 2nd or 3rd condition causes the reduced program to terminate by exception on the test data.

We assume that these restrictions are satisfied by the default mode reordering algorithm of MMC as

reused in Reductor, if strict sequential operational semantics are used. We also believe that the use of any algorithm that performs minimal reordering as specified in Mercury Reference Manual [6] should also satisfy our restrictions. If not, the use of other mode reordering algorithms in MMC might cause problems.

4.4 Collecting data from the deep profiler

Data from the profiler. The MDP consists of two parts: (1) the code inserted by MMC for collecting profiling information into the `Deep.data` file, and (2) a web interface that presents the collected information to the user. We call the information stored in `Deep.data` the **deep profile** and use it in Reductor.

The deep profile contains for each call site in each procedure the list of ports of the standard box model that were used by the procedures called from the call site. More precisely in addition to the standard box model (call, exit, fail, redo), Mercury has one additional port for the procedure throwing an exception.

Transferring data into the IMHLDS. We need to transfer the information about the used ports from the deep profile to the corresponding call sites in IMHLDS.

Unfortunately, the data about call sites from the deep profile do not correspond 1:1 with the HLDS, because the deep profiling code is generated after the construction of HLR (i.e. the semantic analysis or the 4^{th} stage of the compilation). Also the data do not generally contain enough information for an unambiguous pairing of the two call-site structures. We thus had to design a mechanism that accounts for this.

Improving the information on called goals. Because the information from the deep profile that we collected into the HLR is not accurate and it does not give us much information about unifications, we improve our information about which atomic goals are called using the information obtained from MMC semantic analysis.

4.5 Changing the individual items of the program

At this point it is decided which predicate goal will be changed and everything from now on is done locally on individual clauses of the program. We describe the design of the process of transforming each clause of a predicate by constructing term change for a term, given the original term and item that correspond to the clause and HLR with the information about call status of atomic goals.

Transfer of the call status from atomic goals in HLR to atomic goals in ILR. The algorithm for identifying non-removable goals (Section 4.2) operates on the ILR, but we have the information about called goals in the HLR. Therefore, we need to transfer the data to the item level representation.

Also, it should be noted that we reasoned about correctness of the algorithm based on the ILR, but the actual mode reordering algorithm of MMC operates on the HLR. Thus we should make sure that the transition between ILR and HLR does not cause any trouble. Unfortunately this is very technical and we omit it for brevity.

One atomic goal in ILR can have multiple corresponding atomic goals in HLR. We designed an algorithm based on an approximate matching of the atomic goals of the two representations, where the lost information about call status is partially reconstructed. In short, any ILR's atomic goal is marked as called, if there is a called goal in HLR which corresponds to the ILR goal.

The matching of the atomic goals of the two representations is based on the fact that throughout all four stages of compilation we reuse, MMC associates with each goal the information about what source file and line number the goal is located on (for the purposes of error messages). We use this as an (ambiguous) tag of each goal. The matching of the atomic goals of the two representations thus consists of the matching of these tags[3].

Constructing the term change for the clause. As an input to this, for each subgoal of the clause goal (in ILR), we know if the subgoal is removable.

The ILR of a clause corresponds reasonably well to the TeLR of a clause. This fact allows us to construct the term change by simultaneously recursively traversing the clause goal in ILR and its corresponding term while building the term change for the I/O interface. If at any point we are not able to pair the ILR goal with its corresponding term, we just drop the change leaving the problematic term unreduced. This is based on the observation, that changing the removal status of a goal from removable to non-removable is never harmful, we only reduce less.

4.6 Final remarks on dynamical reduction

There remains some unexploited potential for dynamical reduction if we allowed code reordering: reorder for mode constraints *and* maximum dynamical reduction. On the other hand, the implementation of this

[3] Interestingly, the matching of items to terms in I/O interface is based on a similar principle.

extension would be rather difficult and we believe that we would not gain much by this because (1) usually, the programmers write predicates in the "correct" order and (2) if we did major reordering, the reordering might confuse the programmer.

Recently, we learned about the new Profiler Feedback Framework in MMC, which might perhaps have saved us some work. The framework feeds back the information from the deep profile into the MMC. Unfortunately this new and still hidden[4] feature is not available in the version of Mercury the Reductor is based on.

5 Statical reduction

This section describes how Reductor identifies inaccessible procedures to decide which clauses and declarations to remove. As indicated by the name, this analysis is performed statically, without the need to run the program in question.

First, we precisely define the notion of accessible procedures. **Procedure B is accessible from procedure A**, if there is a call site that can call procedure B in a goal of some procedure that is accessible from A. We call **accessible procedure** any procedure in a program that is accessible from the predicate `main`. It is not generally possible to say what procedures can be called from a call site. Reductor thus determines only superset of accessible procedures (elements of the superset may be further called accessible procedures, for brevity).

There are four steps that are taken in the process of statically reducing the procedures of a program:

1. *Calculate the superset of accessible procedures of the program.* Consider the **call graph**, which we define as an oriented graph, where the set of vertices is the set of all procedures in a program and set of oriented edges is defined by relation R. $(A, B) \in R$ if and only if B is called from a call site of A.
   We use a depth-first search on the call graph starting from the predicate `main`. Each procedure that is processed is added to the set of accessible procedures. Processing one node of the search graph consists of finding its call sites (i.e. constructing the edges for the depth-first search). This is done by another depth first search on the goal tree of the procedure, where subgoals are the vertices of the tree and atomic goals (i.e. call goals and unifications) are the leaves of the tree.
   There are two types of call sites: static and dynamic. For static call sites, the procedure to be called is known (in HLR). For dynamic call sites, the procedure is determined at runtime. They are two kinds of a dynamic call site: (a) call sites for lambda expressions, and (b) call sites for instances of a class method.
   Call sites (b) can be considered static, because the class method called is known (the instance is not). Procedures that can be called from (a) are determined by treating the unification that declares the lambda expression as a call site of its lambda expression.
2. *Identify further non-removable procedures.* All accessible procedures are naturally non-removable. Unfortunately, there are cases when even an inaccessible procedure cannot be removed. One example are procedures whose removal would require us to to remove additional declarations (beyond the obvious `pred`, `func` and `mode` declarations and procedure's clauses). Another example are predicates exported from Mercury to the C interface—for these we cannot be sure if they are ever called from some C code.
3. *Recalculate the superset of accessible procedures.* Marking some procedures as non-removable may require the non-removability for further procedures. We use the same depth-first search as above, except that we start in all non-removable procedures instead of `main`.
4. *Remove the items that correspond to the removable procedures.* The removal is done separately for each module of the program through finding their corresponding items and submitting them for deletion to the Reductor's I/O interface.

6 Remarks on the interfacing compilers

We think that in future, Reductor can be extended to be able to remove more declarations and to lift most of the limitations it places on the input program. There are however several areas where we are bounded by the current design of MMC. The most notable thing is that it is very problematic to make changes in the contents of clauses and declarations in the general case, unless we give up the aim of preserving the original formatting of the code, i.e. the major goal of Reductor.

One possible solution would be to extend Reductor's I/O interface so that the reductions could submit changes directly to the HLR instead of just TeLR. Although we did not design such an interface, we believe that it would be reasonable to try to build it, if the MMC had not such an unfavorable design with regard to this proposal. Certainly in the presence of such an interface, the reductions would be quite easy transfor-

[4] For more details see `deep_profiler/feedback.m` in newer versions of MMC source package.

mations of HLR. This would, at the very minimum, save us lot of work on dynamical reduction.

We also believe that this proposed enhanced I/O interface could be easily reused for e.g. automatic code restructuring or refactoring (see e.g. [7]) or intelligent syntax highlighting. These two kinds of software can make a meaningful use of semantic analysis provided by the compiler. The need for such an interface can be illustrated by the fact that the task as simple as function renaming requires type analysis (which is a part of the semantic analysis) to distinguish data constructors of discriminated unions from function calls of the same name and arity.

7 Conclusion

We described two main automatic methods for source code reduction of Mercury programs aimed at understanding and reusing parts of Mercury projects. The statical reduction removes predicates that are never called (based on the statical analysis of the source code). The dynamical reduction removes parts of predicates that were never used on some sample input data. The formal correctness of the code is preserved by inserting exceptions at these places. The dynamically reduced code is guaranteed to return the same outputs on the inputs it was reduced for. On different outputs, the reduced program may exit with an exception.

The implemented tool called Reductor performs both types of the reduction while preserving the original formatting of the source code as much as possible to enable further development. This is a unique feature that complicated the design of Reductor by far the most.

Reductor has been tested on few medium-sized programs and a large set of small programs (mostly from the MMC test suite). The reductions are sufficiently powerful. There are few imposed constraints on the Mercury language for statical reductions which concern rather rare features. Moreover, the presence of such unsupported constructs usually does not prevent Reductor from reducing the program. While in these cases the reduced program may become uncompilable, it is still useful for manual inspection. In the case of dynamical reduction, the tests did not reveal any substantial quantitative degradation in the amount of code reduced by the implemented method as compared to what can be potentially reduced by a similar method without the mentioned approximations.

The utility of Reductor for large scale projects has to be confirmed yet. We believe that use of dynamical reduction[5] on the well chosen subset of all possible input data sets, combined with statical reduction and some subtle changes in the original code, can be useful in understanding and/or reusing code of big programs. This was the main motivation behind building this tool.

We have also commented on some limitations of the design of MMC, briefly explored the possibilities for extending Reductor and gave some thoughts on commonalities of Reductor code with some other development tools like automatic code refactoring and intelligent syntax highlighting.

[5] The dynamical reduction can also be viewed as coverage testing tool that "visualizes" its results by commenting out the unused parts of code.

References

1. Mercury project WWW page. `http://www.mercury.csse.unimelb.edu.au/`
2. Z. Somogyi, F. Henderson and T. Conway: *Mercury: an efficient purely declarative logic programming language.* ASCS'95, Glenelg, Australia, 1995, 499–512.
3. J. Diviš: *Automatické odstranění nadbytečných částí programu*[6]. Bachelor thesis at Charles University in Prague, Faculty of Mathematics and Physics, 2009.
4. Ralph Becket's Mercury tutorial. `http://www.mercury.csse.unimelb.edu.au/tutorial/book/book.pdf`
5. Seriál Mercury on ROOT.CZ. `http://www.root.cz/serialy/mercury/`
6. Mercury Reference Manual, version 0.13.1. `http://www.mercury.csse.unimelb.edu.au/information/doc-release/mercury_ref/`
7. T. Mens and T. Tourwé: A Survey of Software Refactoring, IEEE Transactions on Software Engineering, **30** (2), February 2004.

[6] This is the formal title. The contents are written in English.