A Correct Refactoring Operation to Rename Global Variables in C Programs

Julien Cohen

A Correct Refactoring Operation to Rename Global Variables in C Programs

Julien Cohen
Université de Nantes

Abstract. Most integrated development environments are shipped with refactoring tools. However, their refactoring operations are often known to be unreliable. As a consequence, developers have to test their code after applying an automatic refactoring.

In this report, we consider a refactoring operation (renaming of global variables in C), and we prove that its core implementation preserves the set of possible behaviors of transformed programs. That proof of correctness relies on the operational semantics of C provided by Compcert C in Coq. We also prove some static properties of the considered refactoring operation and we use them to find properties of some composed refactoring operations.

1 Introduction

1.1 Refactoring tools are unreliable

Problem 1: Refactoring tools are not reliable. Designing refactoring tools is a complex task because of the complexity of the underlying programming languages. As a result, probably all refactoring tool suffer from small bugs, that occur in rare cases, but that make those tools unreliable. Many programmers have faced situations where the refactoring tool changed their program in an unexpected way. Recent work have found tens of bugs in several refactoring tools [Mon11, Soa12] by systematic testing. This situation prevent programmers to trust their tools.

There are some refactoring tools, such as the Haskell Refactorer [LTR05] that are rigorously designed and developed, but they are finally not free of bugs.[1]

This make refactoring tools unreliable. It regularly occurs that one triggers a refactoring operation, then either finds its program broken and then has to unroll the transformation, or discovers at test time that the transformation has changed the behavior of the program in an unexpected way.

Problem 2: Composition of refactoring operations is hazardous. Basic refactoring operations are sometimes used as building blocks for more complex transformation. For instance, the refactoring tools for Java in IntelliJ IDEA and Eclipse internally use the operation *pull-up method* to provide the operation *extract superclass*. Another example is the transformations of modular architecture described in [CD11] (with the Haskell Refactorer, 19 operations used) and [CDA12, AC13, ACR13] (with IntelliJ Idea, 37 operations were used). In such composed transformations, a fault in any of the basic operation makes the whole chain impracticable.

Moreover, when one wants to build a static analysis on composed operations (as in [KK04, CA13]), the properties basic refactoring operation cannot be formally verified, which make the whole analysis unsound.

[1] The author of this report reported several bugs in the Haskell Refactorer (2010, 2011) while preparing the work described in [CD11].

1.2 Proving properties of refactoring tool operations

Testing and proving are complementary approaches to software validation. Although the need for safe refactoring tools is recognized [SEdM08, BS15], few efforts have been done in this way. For instance, proofs of correctness of refactoring tools are restricted to small languages, such as [ST08]. To the best of our knowledge, no such work exists for industrial languages.

Formalized refactoring in Compcert C. Our goal in this report is to make a step towards a fully formalized refactoring tool for an industrial language. We choose C language because is has a mechanized formalization : Compcert C [BL09, INR15]. We present in section 2 how we use Compcert C to build a refactoring tool, and the limits of the approach. Then we focus on a refactoring operation: renaming global variables. Renaming seems to be inoffensive at first sight, but we will see that there are some pitfalls to be avoided (Sec. 3.1). We show how our implementation handle several situations of masking (Sec. 3.2) and that it preserved the behavior of programs, as expected for refactoring operations (Sec. 3.3.1). We also give some properties of the operation that can be used for static composition (Sec. 3.3.2 and 3.3.3) and we illustrate this with a toy example of composite operation (Sec. 4)

All the properties we give are proved in Coq. The full code and the proofs are available from the author or the project webpage.

2 Refactoring in Compcert C

The main property we want to prove about refactoring operations is their preservation of the behavior of the transformed program/ To do that we need to have a formal definition of the semantics of programs. We rely on the semantics of C programs formalized in Coq provided by Compcert C, which take into account a subset of ISO C 99 larger than MISRA C 2004[2].

That semantics (module `Csem` in Compcert) is defined on abstract syntax trees (AST), not on source code grammar. For this reason, we focus on AST transformations in this report. This allows to focus on the logic of the transformation, but is also has some limits as described below (Sec. 2.4).

2.1 Compcert C syntax and semantics

2.1.1 Abstract Syntax Trees

Identifiers (`AST.ident`) are represented by integers (`BinNums.positive`). A map from textual identifier in the original program an integer identifiers is maintained during parsing (`intern_string` defined in the OCaml module `Camlcoq`).

Programs (`AST.program`) are represented by the list of their definitions (or declarations) : global variables and functions. A definition is represented by a pair (i, d) where i is the defined identifier and d the content of that definition.

That content is composed of a type and an initialization for a global variable (`AST.globdef`), and of a return type, a list of parameters, a list of local variables and a body for a function (`Csyntax.fundef`). Function bodies are represented by statement ASTs (`Csyntax.statement`). Statement ASTs follow a 13 cases grammar, and rely on ASTs of expressions (`Csyntax.expr`) that follow a 22 cases grammar.

At leaves, we find in particular the construction `Evar (x:ident) (ty:type)` to represent local or global variables or function names.

2.1.2 Formal semantics

The semantics for C programs given in Compcert is based on small-step transitions (relation `Csem.step`). That relation respects the non-determinism of C programs : several transitions can be possible from a given state.

[2]See http://compcert.inria.fr/compcert-C.html#subset for the list of unsupported features.

The transitive closure of `step` is used to define the relation `program_behaves` between programs and their possible behaviors. Behaviors are represented with the following data-type [INR15]:

```
Inductive program_behavior: Type :=
  | Terminates: trace -> int -> program_behavior
  | Diverges: trace -> program_behavior
  | Reacts: traceinf -> program_behavior
  | Goes_wrong: trace -> program_behavior.
```

Those behaviors embed *traces* which are lists of observable events, such as outputs, either finite (`trace`) or infinite (`traceinf`).

2.2 Program Transformations

Our refactoring operations are defined as transformations of syntax trees.

We then use a pretty printer to recover a source file. This is not sufficient for a real-world tool,[3] but we focus here on the "provable" part of the transformation.

A program transformation may fail and return an error (`Error message`) or succeed and return the transformed AST (`OK program`, see the module `Errors` of Compcert).

2.3 Behavior preservation of Transformations

2.3.1 Traces, external behavior

As expected for refactoring operation, we focus on preservation of external or observable behavior. This boils down to preserve the relation `program_behaves`.

Preservation of that relation ensures preservation of the termination, of the returned result, and of the trace.

We accept that the internal behavior (control flow, memory footprint...) change as many existing refactoring operations imply that. The internal behavior is not reflected in the `program_behavior` type, except for some accesses to global variables.

Because of some insight on the internal behavior (global variables) in traces, we have to preserve traces *up to* global variables renaming (next section).

2.3.2 Relaxing trace equality

Compcert traces include references to global variables that are not observable if those variables are not volatile (and if libraries do not use them). Since we aim at renaming global variables, we have to accept that traces change up to some renaming.

For this reason, our refactoring operation comes with an operation that describes how traces are modified.

2.3.3 Non-determinism

We do not only want that the possible behaviors of the transformed program are included in the set of possible behaviors of the initial program, we want it to be the same set.

For instance, consider the two programs below. You can pass from one to another by applying an *extract constant* or *unfold constant* refactoring operation. However, the two programs do not have the same set of possible traces. The first one can only print "AB" whereas the second can print "AB" or "BA".[4]

[3] Other tools (at least IntelliJ Idea, Eclipse and HaRe for Haskell) transform the AST, then reflect the changes in the token stream.

[4] See [Kre14] for other examples of this kind.

```
string a = "A"  ;
string b = "B"  ;

int main(){
  int r1 = printf(a)  ;
  int r2 = printf(b)  ;
  return r1 + r2  ;
  }
```

```
string a = "A"  ;
string b = "B"  ;

int main(){


  return printf(a) + printf(b);
  }
```

This kind of change should be avoided. When this is inherent of the considered refactoring operation, the user should be aware that the behavior is not strictly preserved.

2.3.4 Bisimulation

Bisimulation would require that each reduction/transition step has an equivalent in the two internal behaviors (in the original program and in the transformed program). Bisimulation is not needed. When bisimulation is ensured for a given refactoring operation (such as renaming variables), it can be used to ease the proof of correctness.

2.4 Limits of the approach

Working on ASTs implies the following limits :

- (Pre-processing.) We do not take pre-processing into account. We do not reconstruct pre-processor directives.
- (Lexing and pretty-printing.) We do not preserve layout and comments.
- (Bloc variables.) Compcert syntax represents block variables by function local variables in the AST. Since we consider only ASTs, we have no way to considered different maskings in different blocks of functions. In particular, our renaming operation fails to rename `x` into `y` on the following program whereas it could be legally transformed (*i.e.* without capture).

```
int x = 1 ;

void main(void){
  x++ ;
  {
    int y = 1 ;
    y++ ;
  }
}
  /* renaming x into y is correct here */
```

  Note that this does not affect the correctness of the refactoring operation but it prevents its completeness.

- Some parts of the tool cannot be proven correct in the the same framework and have to be tested.

3 Refactoring Operation: renaming global variables

We now focus on a refactoring operation that renames global variables. Because of many masking and capture situations, renaming is often considered difficult.[5]

[5]Extract from the web page of Microsoft Visual C++ Refactoring (updated 5/2/2015) which is experimental: *"The C++ language is large and complex with context-sensitive syntax subtleties that make it difficult to create a reliable and fast rename refactoring tool. [...] You might still need to manually go through unknown/unconfirmed results proportional to the scope of the refactoring for complex refactors, but for many common scenarios little intervention is required."*

3.1 Specification

3.1.1 Name bindings, masking, captures

The interesting part of renaming variables, of course, is dealing with masking (a local variable *masks* a global variable when they have the same name). We expect the following features :

- We cannot introduce captures (that would change the behavior).
- We can introduce makings as long as they do not create a capture. For instance, renaming x into y in the program below introduces a new masking, but is correct.

```
int x ;
int f(int y){
  return y + 1 ;
}
  /* renaming x into y is correct here */
```

See however the limit explained in Sec. 2.4.

3.1.2 Volatile Variables

We do not rename volatile variables because they are designed to be shared with outside world.

3.1.3 External Functions

The problem. In Compcert C, external function (system calls, library function calls) have a (restricted) access to global variables (see the property `extcall_properties.ec_symbol_preserved` enforced by the axiom `external_functions_properties` in Compcert module `Events`).

This reflects the fact that a global variable can be accessed from a linked library, which source code may not be available. The following program gives such an example:

```
/* This is the main program */

void show(); /* external function declaration,
                defined in a library          */

int a = 0 ; /* global variable definition */

int main(){
  a = 1 ;
  show() ;
  a = 2 ;
  show() ;
}
```

```
/* Library (shipped without source code) */

#include <stdio.h>

extern int a ; /* declaration (not a definition) */

void show(){
  printf ("%d\n", a);
}
```

Sufficient Condition. Since we generally cannot propagate the renaming in libraries, we have to assume that the renamed variable is not accessed from external code.[6] This is formalized by the following predicate

[6]This could be relaxed with an analysis of the external library compiled code, but this is out of the scope of this report.

`extcall_additional_properties` (a conjunction of four conditions) where x represents the old name and y the new name (module `ExtCall`). That predicate is used as a precondition for behavior preservation (Sec. 3.3.1).

```
Definition does_not_exhibit_var (sem:Events.extcall_sem) (x:ident) : Prop :=
  ∀ ge vargs m1 tra vres m2,
    sem fundef Ctypes.type ge vargs m1 tra vres m2 → ¬appears_trace x tra.

---

Record extcall_additional_properties (sem:Events.extcall_sem) x y : Prop :=
  mk_extcall_properties {

      x_not_in_trace : does_not_exhibit_var sem x ;

      y_not_in_trace : does_not_exhibit_var sem y ;

      stability_right :
        ∀ ge tr_ge vargs m1 tra vres m2,
          GlobalEnv.rename_globvar x y ge = OK tr_ge →
          sem fundef Ctypes.type ge vargs m1 tra vres m2 →
          sem fundef Ctypes.type tr_ge vargs m1 tra vres m2 ;

      stability_left :
        ∀ ge tr_ge vargs m1 tra vres m2,
          GlobalEnv.rename_globvar x y ge = OK tr_ge →
          sem fundef Ctypes.type tr_ge vargs m1 tra vres m2 →
          sem fundef Ctypes.type ge vargs m1 tra vres m2
    }.
```

That predicate is used as a pre-condition on the semantics of external functions and for embedded assembly code (see Fig.1). It is not set as an axiom because it need only to be ensured for the two names involved in the renaming.

3.2 Implementation of the transformation

We consider `rename x y` (when $x \neq y$).

Pre-operations. Before calling the transformation defined in Coq we perform these operations (coded in OCaml) :

1. Check that y is not a keyword. This cannot be done in the Coq part because identifiers are represented by numbers in syntax trees.
2. Launch parsing with as few as possible modifications of the AST (the Compcert compiler changes the names of local variables which mask global variables just after parsing but we cannot do the same in order to recover original names in our output).

Top-level checks and transformation. When the AST is available from parsing, the transformation defined in Coq operates as follows (see the module `Programs`, function `rename_globvar_hard`):

1. Check that x and y are different from *main*.
2. Check that y is not already declared (as a global variable or a function).
3. Check that x is not a function.

4. Check that x is not volatile.
5. Change the declaration of x into of declaration for y.
6. Change occurrences of x in global variable initializations into occurrences of y. Fail is x is used in its own initialization.
7. Propagate the change in functions (see below; can fail).

Renaming in function definitions. To propagate a renaming in a function f, we first check if f binds x and y by the means of a parameter or a local variable (see *propagate_change_ident* below).

```
Definition force (x:AST.ident) (y:AST.ident) (f:Csyntax.function) :=

  do s ← Statements.change_ident x y (fn_body f) ;
  OK (mkfunction (fn_return f) (fn_callconv f) (fn_params f) (fn_vars f) s).
```

```
Definition propagate_change_ident
           (x:AST.ident) (y:AST.ident) (f:Csyntax.function) :=

     if dec_binds x f
     then
       if dec_binds y f
       then OK f
       else
         if decidable_appears_statement y (fn_body f)
         then Error (msg "Replacing identifier occurring in function.")
         else OK f

     else
       if dec_binds y f
       then
         if decidable_appears_statement x (fn_body f)
         then Error (msg "This renaming would introduce an undesired hiding.")
         else OK f
       else force x y f
  .
```

Four different situations can occur :

- f does not bind x / f does not bind y : Rename x into y in the body of f (a statement). This will report an error if y is encountered.
- f binds x / f binds y : Do not change the body of f (masking).
- f does not bind x / f binds y : If x appears in the body of f, report an error (capture). Else do not change the body of f.
- f binds x / f does not bind y : Do not change the body of f (masking). Also check that y does not occur in the body of f. Otherwise, we would transform an ill-formed program into a well-formed program, as for the following progam :[7]

[7] For instance, for that program, Visual Assist (Whole Tomato Software) for Visual Studio will perform the renaming of the global x into y, yielding a valid program from an invalid one, which obviously changes the semantics.

```
int x;

int f(int x){
  return y ;     /* this instance of y is free*/
}
```

Statements and Expressions. Renaming is propagated to the `Evar` leaves of AST of Statements and Expressions, returning an error if the new identifier is encountered as a variable name (labels are not checked).

```
Definition change_ident_untyped (x : AST.ident) (y:AST.ident) i :=
          if AST.ident_eq x i
          then OK y
          else
            if AST.ident_eq i y
            then Error (msg "replacing identifier already occurs")
            else OK i .
```

We also report an error in some cases that are avoided by construction (some constructors in the grammar of external calls are introduced only during compilation phases of Compcert C ; this is formalized as a precondition, see `RawGlobals.rawglobals` in Sec. 3.3.1).

3.3 Properties

3.3.1 Behavior preservation

```
Variable x : AST.ident.
Variable y : AST.ident.

Hypothesis EXT1 :
  ∀ name sig,
    ExtCall.extcall_additional_properties (Events.external_functions_sem name sig) x y.
Hypothesis EXT2 :
  ∀ text,
    ExtCall.extcall_additional_properties (Events.inline_assembly_sem text) x y.

Variable p : program.
Hypothesis RG : RawGlobals.rawglobals p .

Theorem behavior_preserved1 :
  x ≠ y →
  ∀ (t_p:program) (b : Behaviors.program_behavior),
    Programs.rename_globvar_hard x y p = OK t_p →
    Behaviors.program_behaves (Csem.semantics p) b →
    ∃ t_b,
      ( Behaviors.Unsafe.rename_globvar x y b = OK t_b ∧
        Behaviors.program_behaves (Csem.semantics t_p) t_b ).
```

Figure 1: Main correctness result

Premises.

- We suppose that external libraries and embedded assembly code does not accesses the renamed variable or do not use the new variable name (hypothesis *EXT1* and *EXT2* in Fig. 1, see Sec. 3.1.3).
- We assume the AST comes directly from the parser and thus does not contain any optimization of accesses to global variables (hypothesis *RG* in Fig. 1).
- We consider the transformation succeeds (*rename_globvar_hard x y p = OK t_p*) and then does not report an error (see Sec. 3.2 and3.3.2 for error cases).

Main Result. Under these conditions, the resulting program has the same behavior up to renaming in the trace. This is formalized by the theorem *behavior_preserved1* (Fig. 1) proved in Coq (in module `Correctness`).

This theorem means that if the original program can produce a trace, the transformed program can produce the same trace, up to renaming in the trace (see Sec. 2.3.2).

Moreover, we have a symmetric result (theorem *behavior_preserved2* in module `Correctness`) that states that if the transformed program can produce a trace, then the original program can produce the same trace (up to renaming).[8]

Those two results allow to say that the original program and the resulting program have the same set of possible behaviors (non-determinism is preserved).

Structure of the Proof. The key lemma to prove the main result it the preservation of transitions (Fig 2).

Renaming does not change the control flow of the program. For this reason, we can exhibit a bisimulation at each level of semantics (expression reduction, see module `Reduction`, state transitions, see module `Step`). To do this, we build a correspondence between states coming from the execution of the initial program with states in the execution of transformed program (function `change_ident` in module `State`). That correspondence relies on correspondences between continuations, between contexts, and between global environments (defined in modules `Continuations`, `Contexts` and `GlobalEnv`).

Difficulties. Those correspondences on continuations, contexts and global environments are tricky to build for different reasons. For continuations, bindings are not always explicit and have to be analyzed before propagating a renaming (as for function bindings). For contexts, Compcert uses an higher order representation which makes context transformation difficult. For global environments, Compcert uses dependant types which forbid the use of some usual tactics in Coq. These points are explained in App. A.

Another technical difficulty is the large number of cases in grammars (types of expressions, statements and continuations) that make quadratic analyses (injectivity proofs for instance) very heavy in time and memory (some tactics take more than 10 minutes at 2 Ghz and more than 4 GB of memory).

Open questions.

- What can we say about ill-formed programs? We only deal with programs for which parsing is successful. What kind of errors does it reject, what kind of errors does it accept?
- What can we say (and prove) about internal behavior (memory usage for instance)?[9]
- What are the cases we reject and that would be legally renamed (as explained in Sec. 2.4) ?

[8] In this result, we use the bisimilarity between infinite traces defined in Compcert (`traceinf_sim`) instead of Coq syntactic equality (see `TraceProperties.behavior_res_eq`).

[9] The memory is unchanged at initialization, see theorem `transforme_init_mem` in module `Memory`.

```
Lemma step_commut :
  ∀ x y ge tr_ge st1 tr_st1 tra tr_tra st2 tr_st2,

    (∀ name sig, ExtCall.extcall_additional_properties (Events.external_functions_sem name sig) x y) →
    (∀ text, ExtCall.extcall_additional_properties (Events.inline_assembly_sem text) x y) →

    GlobalEnv.rename_globvar x y ge = OK tr_ge →
    State.change_ident x y st1 = OK tr_st1 →
    State.change_ident x y st2 = OK tr_st2 →
    rename_in_trace x y tra = OK tr_tra →
    step ge st1 tra st2 →
    RawGlobals.rawglobals_state st1 →
    wf_state st1 →
    step tr_ge tr_st1 tr_tra tr_st2.
```

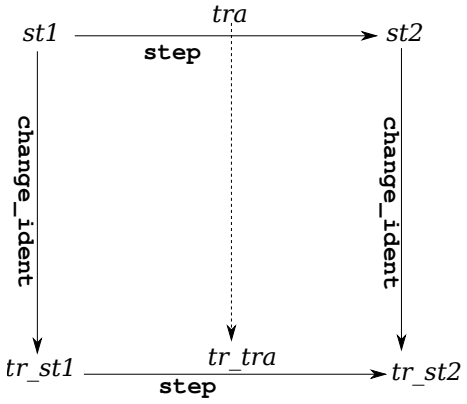

Figure 2: Commutativity of renaming with transitions (relation `step`)

3.3.2 Precondition (for the transformation to succeed)

The transformation fails (returns an error) in some problematic situations where the behavior would not be preserved. Here we describe the conditions on programs that make the transformation succeed.

Necessary preconditions. The following lemmas gives some necessary preconditions.

```
Lemma rename_not_defines_globvar :
  ∀ x y p t_p,
    x≠y →
    rename_globvar_hard x y p = OK t_p →
    ¬ Bindings.defines_globvar_prog y p .
```

That lemma states that the new name must not be bound to a global variable.

```
Lemma rename_not_defines_func :
  ∀ x y p t_p,
    x≠y →
    rename_globvar_hard x y p = OK t_p →
    ¬ Bindings.defines_func_prog y p .
```

That lemma states that the new name must not be bound to a function.

```
Lemma rename_appears_free :
  ∀ x y p t_p,
    rename_globvar_hard x y p = OK t_p →
    ¬ Bindings.appears_free y p
    ∧ ¬ Bindings.appears_free x p .
```

That lemma states that the renaming fails if the new name or the old name appear free in the program.

```
Lemma rename_defines :
  ∀ x y p t_p,
    rename_globvar_hard x y p = OK t_p →
    Bindings.defines_globvar_prog x p.
```

That lemma states that the old name must be bound to a defined global variable. This points a choice in the design/specification of the renaming operation: one can build a renaming operation that successes if the variable to be renamed does not exist. This choice is not important here.

Sufficient preconditions. We introduce the predicate `covers y x f` that characterizes situations where renaming `x` into `y` in a function `f` would introduce a capture.

```
Definition covers y x f :=
  binds y f ∧ ¬binds x f ∧ appears_statement x (fn_body f).
```

The following result shows which conditions are suffient for the ransformation to succeed.

```
Theorem sufficient_precondition :
  ∀ x y p,
    x ≠ y →
    x ≠ prog_main p →
    y ≠ prog_main p →

    RawGlobals.rawglobals p →

    defines_globvar x p →
    ¬defines_globvar y p →
    ¬defines_volatile_globvar x p →

    ¬defines_func x p →
    ¬defines_func y p →

    ¬appears_free y p →
    ¬appears_free x p →
    (∀ (f : function) (i : ident),
        List.In (i, Gfun (Csyntax.Internal f)) (prog_defs p) → ¬covers y x f) →

    ∃ t_p, rename_globvar_hard x y p = OK t_p .
```

The last condition ensures that captures of local variables are avoided whereas other conditions are related to global bindings or good form of programs.

3.3.3 Forward and backward descriptions

We now give some descriptions of the effects of our renaming operation. These descriptions can be used in a static composition calculus. We consider $x \neq y$. We also consider z such that $z \neq x$ and $z \neq y$.

The following forward descriptions are proved for `rename x y p` (and `t_p` refers to the resulting program):

- `defines_globvar` commutes (lemmas *defines_globvar_z*, *defines_globvar_x* and *rename_not_defines_globvar*):

$$\begin{array}{lcl} \texttt{defines\_globvar z p} & \rightarrow & \texttt{defines\_globvar z t\_p} \\ \texttt{defines\_globvar x p} & \rightarrow & \texttt{defines\_globvar y t\_p} \\ \texttt{defines\_globvar y p} & \rightarrow & \bot \end{array}$$

- `defines_func` (lemmas *defines_func_z*, *defines_func_x* and *rename_not_defines_func*):

$$\begin{array}{lcl} \texttt{defines\_func z p} & \rightarrow & \texttt{defines\_func z t\_p} \\ \texttt{defines\_func x p} & \rightarrow & \bot \\ \texttt{defines\_func y p} & \rightarrow & \bot \end{array}$$

- `appears_free` (lemmas *appears_free_z* and *rename_appears_free*):

$$\begin{array}{lcl} \texttt{appears\_free z p} & \rightarrow & \texttt{appears\_free z t\_p} \\ \texttt{appears\_free x p} & \rightarrow & \bot \\ \texttt{appears\_free y p} & \rightarrow & \bot \end{array}$$

- `RawGlobals` stable (lemma *rg_stable*):

$$\texttt{raw\_globals p} \rightarrow \texttt{raw\_globals t\_p}$$

The following backward descriptions (properties needed on `p` are deduced from properties desired on `t_p`) are also proved for `rename x y p` :

- `defines_globvar` commutes (lemmas *defines_globvar_z_back*, *rename_defines_y* and *rename_not_defines_x*):

$$\begin{array}{lcl} \texttt{defines\_globvar z t\_p} & \rightarrow & \texttt{defines\_globvar z p} \\ \texttt{defines\_globvar y t\_p} & \rightarrow & \top \\ \texttt{defines\_globvar x t\_p} & \rightarrow & \bot \end{array}$$

- `defines_func` is stable (lemmas *defines_func_z_back*, *not_defines_y_func*, *not_defines_x_func*):

$$\begin{array}{lcl} \texttt{defines\_func z t\_p} & \rightarrow & \texttt{defines\_func z p} \\ \texttt{defines\_func y t\_p} & \rightarrow & \bot \\ \texttt{defines\_func x t\_p} & \rightarrow & \bot \end{array}$$

- `appears_free` (lemmas *appears_free_z_back*, *rename_appears_free2_x* and *rename_appears_free2_y*):

$$\begin{array}{lcl} \texttt{appears\_free z t\_p} & \rightarrow & \texttt{appears\_free z p} \\ \texttt{appears\_free x t\_p} & \rightarrow & \bot \\ \texttt{appears\_free y t\_p} & \rightarrow & \bot \end{array}$$

- `RawGlobals` stable (lemma *rg_backstable*):

$$\texttt{raw\_globals t\_p} \rightarrow \texttt{raw\_globals p}$$

4 Static Composition

In this section, we illustrate static composition by giving some properties of composed refactoring operations from the properties of their components.

We suppose we want a refactoring operation that switches the names of two variables. We are going to build that operation by using the simple renaming operation as a black box.

4.1 A Naive attempt

We first consider the following implementations.

```
Definition swap0 x y p :=
  do p1 ← rename_globvar_hard x y p ; rename_globvar_hard y x p1.
```

We find that y cannot appear in the resulting program (lemma *defines_globvar_back* below), which obviously violates the intuitive specification of the operation.

```
Lemma defines_globvar_back :
  ∀ x y p t_p,
    x ≠ y →
  swap0 x y p = OK t_p →
  defines_globvar y t_p → False.
Proof.
  intros x y p t_p D T.
  monadInv T.
  apply rename_not_defines_x in EQ0 ; auto.
Qed.
```

Note that this result is obtained by using only properties given in section 3.3.3. The base component operation is considered as a black box : we do not inspect its code and the analysis is static : we never run the component.

4.2 A better Swap operation

We now consider a second implementation :

```
Definition swap x y tmp p :=
  do p1 ← rename_globvar_hard x tmp p ;
  do p2 ← rename_globvar_hard y x p1 ;
  do p3 <- rename_globvar_hard tmp y p2 ;
  OK p3.
```

The following forward and backward descriptions hold (with x, y, *tmp*, and z all different).[10]

- Forward descriptions (lemmas `defines_x`, `defines_y`, `not_defines_tmp` and `defines_globvar_z` in module `Swap`):

$$\begin{aligned} \top &\rightarrow \texttt{defines\_globvar x t\_p} \\ \top &\rightarrow \texttt{defines\_globvar y t\_p} \\ \texttt{defines\_globvar tmp p} &\rightarrow \bot \\ \texttt{defines\_globvar z p} &\rightarrow \texttt{defines\_globvar z t\_p} \end{aligned}$$

[10] As for all forward and backward descriptions, the corresponding lemmas consider the transformation has succeeded. One must use these properties in conjunction with preconditions to build the precondition of a composed operation [CA13].

- Backward descriptions (lemmas `defines_x`, `defines_y` (same lemmas as in forward description), `not_defines_tmp_back` and `defines_globvar_z_back` in module `Swap`):

$$\begin{aligned}
\texttt{defines_globvar x t_p} &\rightarrow \top \\
\texttt{defines_globvar y t_p} &\rightarrow \top \\
\texttt{defines_globvar tmp t_p} &\rightarrow \bot \\
\texttt{defines_globvar z t_p} &\rightarrow \texttt{defines_globvar z t p}
\end{aligned}$$

5 Conclusion

5.1 Results

We have built a refactoring tool which logic part is formally described and proved correct. Our tool produces C code that has the same semantics as the initial program. The layout and the preprocessing directives are not preserved in our prototype. Also, separate compilation is not fully supported in our prototype since we do not check the use of the renamed variable in linked libraries.

5.2 Related work

- On the formalization of refactoring operations on a core language: [LT05] and [ST08] (checked by Isabelle/HOL).
- On automatic testing of refactoring operations [Mon11, Soa12], and automatic testing of refactored programs [MGS+14, GMH14].
- Using refactoring for formal verification [YKNW08] [YKW09], which also needs verified refactoring operations.

5.3 Future Work

Validation of widespread refactoring tools. Proving the correctness of refactoring operations made us point some situations that require a deep understanding of C mechanisms. For instance, the fact that any library has a direct access to all global variables of the program (see section 3.1.3) can be unexpected by some programmers. That experience can be used to provide a set a tricky test cases for IDE refactoring tools. As an example, we have found a case that is not well treated in a largely used tool (see footnote 7).

Refactoring ill-formed programs. Some refactoring tools can perform correct refactoring in ill-formed programs, for example when there is a syntax error in a part of the program that is not concerned with a local change. Refactoring such programs is not well covered in our work.

Preserve the layout, preserve the preprocessing directives. Most refactoring tools preserve the layout of programs and our tool could probably adapted with the techniques they use.

Preserving pre-processing directives is probably more difficult. This is studied in [Spi03, Vit03, Gar05, Pad09, Spi10].

Separate compilation. Our tool should take into account all the files in the project (other C files, libraries) to be effective. Another possibility is to produce a patch for parts of the project that cannot be inspected or modified, as [HD05] does.

Other refactoring operations. Of course, a large set of popular refactoring operations either atomic or composed are waiting to be verified.

Combine test and proof. Currently, only the AST transformation is proved correct. We use tests to check that the parser and the pretty-printer correctly interact with our AST transformation. We could study which tests are necessary to complement our proof.

References

[AC13] Akram Ajouli and Julien Cohen. Refactoring Composite to Visitor and Inverse Transformation in Java. Technical report, July 2013.

[ACR13] A. Ajouli, J. Cohen, and J.-C. Royer. Transformations between composite and visitor implementations in java. In *Software Engineering and Advanced Applications (SEAA), 2013 39th EUROMICRO Conference on*, pages 25–32, Sept 2013.

[BL09] Sandrine Blazy and Xavier Leroy. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 43(3):263–288, 2009.

[BS15] J. Brant and F. Steimann. Refactoring tools are trustworthy enough and trust must be earned. *Software, IEEE*, 32(6):80–83, Nov 2015.

[CA13] Julien Cohen and Akram Ajouli. Practical use of static composition of refactoring operations. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 1700–1705. ACM, 2013.

[CD11] Julien Cohen and Rémi Douence. Views, Program Transformations, and the Evolutivity Problem in a Functional Language. 19 pages, January 2011.

[CDA12] J. Cohen, R. Douence, and A. Ajouli. Invertible program restructurings for continuing modular maintenance. In *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, pages 347–352, March 2012.

[Gar05] A. Garrido. *Program refactoring in the presence of preprocessor directives*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 2005.

[GMH14] Xi Ge and Emerson Murphy-Hill. Manual refactoring changes with automated refactoring validation. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1095–1105, New York, NY, USA, 2014. ACM.

[HD05] Johannes Henkel and Amer Diwan. Catchup!: Capturing and replaying refactorings to support api evolution. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 274–283, New York, NY, USA, 2005. ACM.

[INR15] INRIA. Compcert C web page : http://compcert.inria.fr/compcert-C.html, 2007–2015.

[KK04] Günter Kniesel and Helge Koch. Static composition of refactorings. *Science of Computer Programming*, 52(Issues 1-3):9–51, 2004. Special Issue on Program Transformation.

[Kre14] Robbert Krebbers. An operational and axiomatic semantics for non-determinism and sequence points in c. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 101–112. ACM, 2014.

[Ler09] Xavier Leroy. Programming with dependent types: passing fad or useful tool? Informal minutes of IFIP Working Group 2.8 26th meeting, June 2009.

[LT05] Huiqing Li and Simon Thompson. Formalisation of Haskell Refactorings. In Marko van Eekelen and Kevin Hammond, editors, *Trends in Functional Programming*, September 2005.

[LTR05] Huiqing Li, Simon Thompson, and Claus Reinke. The haskell refactorer, hare, and its api. In *Proceedings of the Fifth Workshop on Language Descriptions, Tools, and Applications (LDTA 2005)*, volume 141 of *Electronic Notes in Theoretical Computer Science*, pages 29–34. Elsevier, 2005.

[MGS+14] Melina Mongiovi, Rohit Gheyi, Gustavo Soares, Leopoldo Teixeira, and Paulo Borba. Making refactoring safer through impact analysis. *Sci. Comput. Program.*, 93:39–64, November 2014.

[Mon11] Melina Mongiovi. Safira: A tool for evaluating behavior preservation. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '11, pages 213–214, New York, NY, USA, 2011. ACM.

[Pad09] Yoann Padioleau. Parsing c/c++ code without pre-processing. In Oege de Moor and MichaelI. Schwartzbach, editors, *Compiler Construction*, volume 5501 of *Lecture Notes in Computer Science*, pages 109–125. Springer Berlin Heidelberg, 2009.

[PE88] F. Pfenning and C. Elliott. Higher-order abstract syntax. *SIGPLAN Not.*, 23(7):199–208, June 1988.

[SEdM08] Max Schäfer, Torbjörn Ekman, and Oege de Moor. Challenge proposal: Verification of refactorings. In *Proceedings of the 3rd Workshop on Programming Languages Meets Program Verification*, PLPV '09, pages 67–72, New York, NY, USA, 2008. ACM.

[Soa12] Gustavo Soares. Automated behavioral testing of refactoring engines. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, pages 105–106, New York, NY, USA, 2012. ACM.

[Spi03] D. Spinellis. Global analysis and transformations in preprocessed languages. *Software Engineering, IEEE Transactions on*, 29(11):1019–1030, Nov 2003.

[Spi10] Diomidis Spinellis. Cscout: A refactoring browser for c. *Science of Computer Programming*, 75(4):216–231, April 2010.

[ST08] Nik Sultana and Simon Thompson. Mechanical Verification of Refactorings. In *Partial Evaluation and Program Manipulation*. ACM, Jan. 2008.

[Vit03] Marian Vittek. Refactoring browser with preprocessor. In *Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*, CSMR '03, pages 101–, Washington, DC, USA, 2003. IEEE Computer Society.

[YKNW08] Xiang Yin, JohnC. Knight, ElisabethA. Nguyen, and Westley Weimer. Formal verification by reverse synthesis. In Michael D. Harrison and Mark-Alexander Sujan, editors, *Computer Safety, Reliability, and Security*, volume 5219 of *Lecture Notes in Computer Science*, pages 305–319. Springer Berlin Heidelberg, 2008.

[YKW09] Xiang Yin, J. Knight, and W. Weimer. Exploiting refactoring in formal verification. In *Dependable Systems Networks, 2009. DSN '09. IEEE/IFIP International Conference on*, pages 53–62, June 2009.

A Non-trivial aspects of the proof

We detail here some technical aspects of the proof.

A.1 Compcert State Invariants

We have identified five properties which are true on the continuations embedded in initial states (of the execution of the program) and that are preserved by transitions between states. Those invariants were needed to prove some properties on transformations of contexts and continuations.

- Local environments which are found in continuations are represented by trees which are in a canonical form (property `PTreeProperties.canonique`).

  See `Invariants.wf_canonique_invariant`.

- In continuations, `Kcall` constructions, which contain a function f and a local environment e, a variable is bound by f if and only if it is referenced in e (property `StateProperties.correspond`).

  See `Invariants.wf_locals_invariant`.

- Continuations do not have two consecutive `Kdo` constructors (property `StateProperties.no_kdo`).

  See `Invariants.wf_kdo_invariant`.

- Contexts (represented by Coq functions) which are found in continuations are well formed with respect to the predicate `context` defined in C semantics (module `Csem`).

  See `Invariants.wf_context_invariant`.

- In continuations, `Kcall` constructions, which contain a function f, a context c and a continuation k, the first segment of k and the context c are directly related to the body of f (properties `StateProperties.correspond_context` and `StateProperties.correspond_first_segment`).

  See `Invariants.wappears_f_e_invariant`.

A.2 Higher Order Contexts

The concept of context is familiar in programming language semantics. It is used to specify places where computations can occur. In Compcert, contexts are represented by native Coq functions (type $expr \rightarrow expr$), adopting the higher-order abstract syntax style [PE88]. Higher order data-structures have the well-known drawback of being difficult to inspect and transform.

To transform contexts, we flatten them by applying them to a witness/dummy value, which gives an expression, then we transform that expression, and then we build a function by embedding a substitution mechanism where the dummy values has the role of the placeholder for the formal parameter (see the module `Contexts`).

As a result, proofs on context transformations are more difficult to establish than proofs on transformations of "flat" structures such as expressions.

A.3 Bindings in Continuations

Bindings in continuations are more subtle than bindings in functions. Compcert continuations are "homomorphic" with lists (each continuation constructor has 1 or 0 continuation as parameter).

One of the constructors of continuations, `Kcall`, contains a function. That function binds parameters in all the continuation following that `Kcall` until the next `Kcall`. We call *segment* a part of a continuation delimited by `Kcall` constructors. So the bindings in a segment is fully determined by the bindings of the function in the `Kcall` constructor at the beginning of that segment.

However, the first segment of a continuation may not begin with a `Kcall` constructor. In this situation, the continuation does not contains enough information to determine the bindings in that first segment. The information for that segment is found elsewhere, for instance in the `State` construction that embeds that continuation.

For this reason, we provide two (mutually recursive) transformation functions for continuations : one that propagates the renaming on segments where the variable to be renamed is not masked (`Continuations.change_ident`) and one for the other case, where nothing is transformed until the next segment (`Continuations.skip`).

A.4 Co-induction for infinite traces

Co-induction can be used to prove some (co-inductive) properties on infinite traces (type `Events.traceinf`). However, some properties that are usually proved by induction on finite traces/lists cannot be proved by co-induction on infinite traces/lists. This is the case for instance for the negation of other properties, or for equality. That makes proofs on infinite traces more difficult to establish than those on finite traces (see the module `TraceInf`).

A.5 Global environments as a dependant type

Global environments are implemented in Compcert by structures that embed at the same time plain data (trees) and Coq proofs of properties on those trees (in the style described in [Ler09]). That makes writing transformations of global environment in Coq more heavy that in the traditional way with separated data structures and properties. In particular, some usual Coq tactics such as rewriting in the goal become unavailable for fundamental reasons.